Titre: Title:	Algorithmes et architectures pour l'implémentation de la détection d'expressions régulières
Auteur: Author:	Thomas Luinaud
Date:	2017
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Luinaud, T. (2017). Algorithmes et architectures pour l'implémentation de la détection d'expressions régulières [Master's thesis, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/2707/

UNIVERSITÉ DE MONTRÉAL

ALGORITHMES ET ARCHITECTURES POUR L'IMPLÉMENTATION DE LA DÉTECTION D'EXPRESSIONS RÉGULIÈRES

THOMAS LUINAUD
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
AOÛT 2017

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

ALGORITHMES ET ARCHITECTURES POUR L'IMPLÉMENTATION DE LA DÉTECTION D'EXPRESSIONS RÉGULIÈRES

présenté par : LUINAUD Thomas
en vue de l'obtention du diplôme de : Maîtrise ès sciences appliquées
a été dûment accepté par le jury d'examen constitué de :

M. MARTEL Sylvain, Ph. D., président
M. LANGLOIS J.M. Pierre, Ph. D., membre et directeur de recherche
M. SAVARIA Yvon, Ph. D., membre et codirecteur de recherche
M. DAVID Jean-Pierre, Ph. D., membre

DÉDICACE

À mes parents et grands-parents pour leur soutien,
à Thierry décédé trop tôt,
à ma copine Laurie qui me soutient chaque jour.

REMERCIEMENTS

J'aimerais remercier mon directeur de recherche, Pierre Langlois, ainsi que mon codirecteur, Yvon Savaria, de m'avoir soutenu et aidé tout au long de ce projet en m'apportant un soutien autant technique que moral. Je les remercie également pour le suivi qu'ils ont remarquablement bien effectué tout le long de ma maîtrise. Je voudrais également remercier tous les membres du laboratoire pour les discussions, les échanges d'idées, mais également les moments de détente. Je remercie aussi tous mes proches qui m'ont soutenu tout le long de ce projet malgré la distance. Et j'ai une pensée toute particulière pour mes grands-parents qui sont encore là pour m'accompagner dans mes projets.

RÉSUMÉ

La prochaine génération de réseau mobile, la 5G, devrait supporter des latences 10 fois plus faibles avec des débits et un nombre d'appareils connectés 100 fois plus importants qu'aujourd'hui. Dans le même temps, les opérateurs et les gestionnaires de réseaux veulent des systèmes plus modulaires qui puissent s'adapter rapidement aux nouveaux protocoles, mais qui ne consomment pas plus d'énergie que les solutions actuelles. Les opérateurs et administrateurs sont donc de plus en plus intéressés par des plateformes reconfigurables telles que des FPGA. Cependant, ces plateformes nécessitent encore des experts pour être utilisées et ont des temps de développement qui peuvent être longs ce qui les rend difficiles à intégrer.

De plus, les infrastructures informatiques sont des éléments de plus en plus critiques pour le fonctionnement de l'économie. La sécurité des réseaux est donc devenue un point important pour protéger ces infrastructures. Actuellement la protection des réseaux est effectuée en utilisant des Systèmes de Détection d'Intrusions — *Intrusion Detection System* (IDS) qui effectuent l'inspection en profondeur de paquets — *Deep Packet Inspection* (DPI). Pour permettre la protection, les IDS comparent le contenu des paquets transitant sur le réseau à des règles prédéterminées. Ces règles sont représentées soit par des chaînes de caractères ou bien des expressions régulières.

Dans ce mémoire, nous proposons trois contributions en rapport à l'utilisation de FPGA pour effectuer de la recherche de texte et d'expressions régulières dans les réseaux. Ces trois réalisations sont implémentées sur des FPGA et respectent les contraintes de latence liées aux réseaux.

La première contribution permet de faire de la recherche de chaînes de caractères. Elle implémente des automates séparés en bits. Cette architecture est compatible avec un algorithme de groupement ayant une très faible empreinte mémoire. Les deux éléments combinés permettent de supporter un grand nombre de chaînes de caractères tout en ayant un temps de traitement déterministe. Comme le système est implémenté sur un FPGA, le système peut être reconfiguré pour supporter d'autres tables de règles.

La seconde contribution est un compilateur d'expressions régulières vers le langage VHDL. Ce compilateur permet de simplifier l'implémentation d'expressions régulières sur un FPGA et ainsi de rendre l'utilisation du FPGA plus accessible. En outre, cette réalisation permet le parcours d'automates finis non déterministes — *Nondeterministic Finite Automaton*

(NFA) dans un temps déterministe. Cela permet d'éviter l'utilisation des automates finis déterministes (AFD) qui peuvent nécessiter un nombre d'états exponentiels pour la recherche d'expressions régulières.

Finalement, la troisième contribution est une architecture intermédiaire pour la recherche d'expressions régulières. Cette réalisation permet de traiter deux sujets liés à la recherche d'expressions régulières pour les IDS : la gestion des mises à jour et l'utilisation du préfiltre. L'architecture est multi-contextes et à gros grains. L'utilisation d'une architecture à gros grains permet de rendre le temps de mise à jour d'un contexte plus rapide, car moins de bits de configuration sont nécessaires. Comme l'architecture est multi-contextes, il est possible de traiter le préfiltre en sélectionnant l'expression régulière à chercher, et ce en un seul cycle d'horloge.

ABSTRACT

The next generation of mobile networks, called 5G, is expected to achieve significantly better performance than present networks : latency 10x smaller, throughput 100x higher with 100x more connected devices over the so-called 4G. Moreover, service providers and network administrators will need more configurable systems able to rapidly support new protocols. Furthermore, the power consumption of the resulting network infrastructure remains a critical consideration. A possible solution to meet all those requirements involves the use of FPGAs. However, the development complexity causes integration difficulties.

In addition, computers and data-centers are more and more critical systems. Consequently, security is an important issue. This motivates introducing Intrusion Detection Systems (IDS), which perform Deep Packet Inspection (DPI). IDSs compare the network flow against a set of rules that are expressed with strings and regular expressions.

This thesis proposes three contributions in regard to FPGAs utilization for text and regular expression search. Those contributions respect the latency constraint of networks and are implemented into FPGAs.

The first contribution is a means to perform string matching. It implements a bit-split finite state machine. Also, the resulting architecture is compatible with a low memory consumption grouping algorithm. Combining the grouping algorithm and the architecture allows the support of many strings. Furthermore, thanks to the FPGA, the systems can be adapted to support other sets of rules.

The second contribution is a regular expression-to-VHDL compiler. This compiler simplifies implementation of Nondeterministic Finite Automata (NFA) generated from the regular expression. Moreover, since the implementation is done on an FPGA, the traversal of the NFA is deterministic in time. Subsequently, it avoids the use of Deterministic Finite Automata (DFA) which can require an exponential number of states for regular expressions search.

Finally, the third contribution is a multi-context Coarse Grained Reconfigurable Architecture (CGRA) overlay for regular expression search. This overlay meets two constraints for regular expression search in IDSs : rules update and the use of an integrated filter. Since the overlay is a CGRA, rules update is faster since fewer bits are required to reconfigure a context. The IDS filter consists of an exact string match that indicates which regular expression has to be searched. Because the architecture is a multi-context CGRA, it is possible to change the

context in a single clock cycle. Consequently, it is possible to use the integrated filter of IDSs. In addition, the overlay allows the system to evolve for further need.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

AFD	Automate Fini Déterministe
ASCII	American Standard Code for Information Interchange
ASIC	Application Specific Integrated Circuit, circuit intégré développé pour un client
BM	Bloc Mémoire
BS-FSM	Bit-Split Finite State Machine, machine à états fini divisée en bits
CLB	Configurable Logic Block, bloc de logique programmable
DoS	Denial of Servise, Déni de service
DP-SRAM	Dual Port Static Random Access Memory, mémoire statique à accès direct à double port
EBNF	Extended Backus-Naur Form, forme étendue de Backus-Naur
EP	Élément Programmable
ERDE	Élément Reconfigurable de Détermination d'État
FPGA	Fiel-Programmable Gate Array, circuit intégré prédiffusé programmable
FSM	Finite State Machine, Machine à état fini
GPU	Graphics Processing Unit, processeur graphique
IDS	Intrusion Detection System, système de détection d'intrusions
IPP	Inspection en Profondeur de Paquets
LUT	Look-Up Table, table de correspondance
NFA	Non-deterministic Finite Automata, automate fini non déterministe
OSI	Open Systems Interconnection, interconnexion des systèmes ouvert
PCRE	Perl Compatible Regular Expression, expression régulière compatible avec Perl
PMU	Pattern Matching Unit, unité de comparaison de motifs
PR	Processeurs Réseaux
Regex	Expression régulière
SDN	Software Defined Networking, réseau défini par logiciel
VHDL	Very High Speed Integrated Circuit Hardware Description Language

CHAPITRE 1 INTRODUCTION

En 1989, Tim Berners-Lee invente le web, au CERN, pour permettre l'échange d'informations entre des laboratoires à travers le monde [1]. À la même époque on estime à 100 000 le nombre d'ordinateurs interconnectés. Aujourd'hui, l'internet des objets est dans tous les esprits, et on parle de milliards d'équipements connectés. Tous ces équipements vont donc générer une grande quantité de données qui transiteront sur le réseau et requérir beaucoup de centres de données [2]. Cependant, les centres de données sont aujourd'hui coûteux [3]. Gérer ce trafic sera donc très coûteux, il est donc important de proposer des solutions permettant de réduire ce coût.

1.1 Contexte

La prochaine génération de réseaux, la 5G, devrait être capable de supporter des débits 100 fois supérieur à la 4G, supporter 100 fois plus d'appareils qu'aujourd'hui et avoir une latence 10 fois plus faible qu'actuellement [4]. D'autre part, les Réseaux définis par logiciel — *Software-Defined Networking* (SDN) [5] et le langage P4 [6] rendent la gestion des équipements plus simple et permettent l'intégration plus rapide de nouveaux protocoles. Ceci impose donc les prochains systèmes à être, si ce n'est reconfigurables, capables à minima d'être reprogrammables.

Par ailleurs, le 12 décembre 2015 lors de la COP21, l'accord de Paris a été signé avec pour objectif de limiter à deux degrés Celsius le réchauffement climatique à l'horizon 2100 [7]. Un axe d'action pour tenir cet objectif consiste à réduire la consommation d'énergie. En 2012, 330 TWh d'électricité était consommés par les réseaux de communication dont 77% par les opérateurs de télécommunications [8]. Les centres de traitement de données quant à eux représentent entre 1,1% et 1,5% de l'électricité mondiale consommée en 2010 [9]. Un des objectifs majeur des nouveaux systèmes informatiques est donc de consommer moins d'énergie qu'actuellement.

Dans le même temps, les industries sont de plus en plus dépendantes et sensibles à la sécurité de leurs systèmes d'information [10]. Une solution consiste à analyser tout le trafic entrant le réseau de ces dernières ; les applications faisant cela sont appelées des Systèmes de Détection d'Intrusions — *Intrusion Detection System* (IDS). Les IDS requièrent de traiter une grande quantité de données. Ils doivent également être résilients aux attaques et notamment aux attaques de déni de service — *Denial of Service* (DoS) [11]. Cela implique que ces systèmes

aient un temps de traitement indépendant des données d'entrée mais aussi en accord avec la vitesse du réseau.

Un exemple d'IDS est Snort [12]. Il s'agit d'un logiciel libre permettant d'effectuer l'analyse du trafic sur un réseau en temps réel. Lors de l'analyse, si le paquet est considéré valide alors il va directement à la destination. Si le paquet n'est pas valide alors une action est prise et ces actions peuvent être variées. Par exemple, il peut s'agir de générer une alerte mais également de rejeter le paquet. Cette analyse peut se décomposer en deux parties. La première partie, réalisée par des experts en sécurité, consiste à créer des règles pour déterminer quand les paquets sont invalides. La seconde partie consiste à comparer le trafic à ces règles.

Une règle type de Snort contient deux éléments majeurs, des conditions sur l'entête du paquet et sur le contenu de ce dernier. L'analyse du contenu n'est faite que si la condition sur l'entête est validée. Le contenu est représenté à la fois par des chaînes de caractères et par des expressions régulières. Il est également possible que celui-ci soit vide lorsque qu'il ne doit pas être considéré. Ce contenu s'applique à la couche application du modèle Open Systems Interconnection (OSI) [13].

Le travail présenté dans ce mémoire s'intéresse à la recherche de règles dans le contenu des paquets réseau.

1.2 Éléments de la problématique et motivation

La prochaine génération de réseaux permettra de supporter de plus en plus d'applications. Il est prévu que le nombre d'éléments connectés au réseau double d'ici à 2020 [2]. Dans ces nouveaux objets, une part importante proviendra de l'internet des objets. Dû à leur contrainte d'énergie, les objets connectés délégueront sans doute aux centres de traitements de données les calculs complexes. La sécurité des systèmes effectuant ces calculs devient un sujet de plus en plus important pour les entreprises utilisant des objets connectés [14].

Dans le même temps, les systèmes d'informations sont utilisés pour des applications comme la téléphonie ou la visioconférence qui requièrent des latences faibles. Les administrateurs de réseaux ont donc besoin de solutions pour protéger leurs réseaux. Ces solutions doivent assurer des latences faibles et doivent supporter des mises à jour récurrentes. Les circuits intégrés prédiffusés programmables — *Field-Programmable Gate Array* (FPGA) commencent à être intégrés dans les centres de traitement de données, par exemple chez Microsoft [15]. Les fabricants de FPGA proposent également des solutions pour les centres de traitement, notamment Intel qui a ajouté une composante FPGA à ses processeurs à destination des centres de traitement de données [16].

Cependant, les FPGA sont des systèmes complexes à utiliser [15]. La simplification de la génération des circuits est donc un axe majeur pour permettre de simplifier l'utilisation des FPGA. Dans ce mémoire, l'utilisation d'architectures spécialisées sur FPGA est étudiée. Trois axes sont explorés. Le premier est traditionnel : une architecture spécialisée est proposée. Le second axe explore la génération automatique de circuits à l'aide d'un compilateur. Le troisième axe s'intéresse à la réalisation de circuits plus facilement reconfigurables.

1.3 Objectifs de recherche

Ce mémoire a pour objectif premier de proposer des architectures permettant de réaliser le traitement de donnés au sein des IDS. Trois réalisations sont proposées dans ce document, une implémentant un automate fini déterministe (AFD), et les deux autres implémentent des automates finis non déterministes — *Nondeterministic Finite Automaton* (NFA). La contrainte première considérée pour chaque réalisation est le déterminisme dans le temps de traitement des données. La capacité de mise à jour des différents systèmes est le deuxième point qui est étudié. Finalement, le troisième objectif est de proposer des solutions afin de rendre l'utilisation de matériels spécialisés plus simple. Cela implique à la fois de proposer des architectures standardisées, mais aussi des méthodes pour déterminer la faisabilité de l'implémentation de celle-ci.

1.4 Résumé des contributions

Ce mémoire contient trois contributions principales.

Une première contribution est la réalisation d'une architecture permettant d'utiliser un algorithme de groupement réduisant l'utilisation mémoire pour la recherche de chaînes de caractères. L'architecture réalisée est paramétrable en fonction des éléments générés en sortie de l'algorithme de groupement.

Une seconde contribution est la réalisation d'un compilateur d'expressions régulières vers VHDL. Cette réalisation propose un processus d'automatisation pour l'implémentation et le test d'architectures directement compilées.

La troisième contribution est la réalisation d'une architecture intermédiaire. Cette contribution se décompose en trois éléments. Un premier élément est l'exploration des possibilités

amenées par ce nouveau type d'architecture ainsi que deux types de blocs. Un second élément de la contribution est la proposition de méthodes pour modéliser le coût de l'architecture avant synthèse. Le troisième élément de contribution de cette architecture est un article publié à la conférence ACM Great Lakes Symposium on VLSI 2017 (GLSVLSI'17) [17].

1.5 Organisation du mémoire

La suite de ce mémoire s'organise comme suit. Tout d'abord, des éléments théoriques et une revue des travaux passés sont présentés dans le Chapitre 2. Par la suite, une architecture implémentant un automate fini déterministe (AFD) est exposée dans le Chapitre 3. Le Chapitre 4 présentera un compilateur d'expressions régulières vers le langage VHDL pour une implémentation sur un FPGA. Une architecture intermédiaire reconfigurable sera ensuite abordée dans le Chapitre 5. Finalement, le Chapitre 6 conclut ce mémoire et propose des travaux futurs.

CHAPITRE 2 PRÉLIMINAIRES

Ce chapitre présente les éléments préliminaires concernant l'inspection en profondeur de paquets. La première section présente les éléments de base pour la compréhension du mémoire. La seconde section présente des travaux pour la recherche de texte. Dans une troisième section, les travaux pour recherche d'expressions régulières sont développés. La quatrième section présente les architectures intermédiaires. Enfin, la cinquième section donne une vue d'ensemble de la problématique du projet.

2.1 Définitions et concepts de base

Cette section pose les bases de différents concepts utilisés dans ce document. Dans un premier temps, l'inspection en profondeur de paquets est présentée. Dans une deuxième section, les expressions régulières sont exposées. La troisième section s'intéresse aux compilateurs. Enfin, la quatrième section présente deux types d'automates.

2.1.1 L'inspection en profondeur de paquets

L'inspection en profondeur de paquets — *Deep Packet Inspection* (DPI) consiste à analyser les données passant sur le réseau. Contrairement aux autres systèmes qui s'intéressent uniquement aux en-têtes des paquets, la DPI traite le paquet dans son ensemble. Cette inspection peut être utilisée pour garantir une qualité de service, accumuler des statistiques ou encore pour protéger le réseau. Les systèmes effectuant la protection d'un réseau sont des IDS tel que Snort [12] et Bro [11]. Les IDS fonctionnent en comparant le trafic entrant à des règles prédéterminées.

La figure 2.1 présente un exemple d'une règle pour Snort. Cette règle indique que pour tout paquet `tcp` ne provenant pas du réseau `192.168.1.0/24` et voulant contacter sur le port `11` un des systèmes de ce réseau, et dont le contenu valide l'expression régulière `VRAI+MENT MECHANT`, alors il faut envoyer un message `PAQUET VRAIMENT MECHANT` à l'administrateur.

```
alert tcp !192.168.1.0/24 any -> 192.168.1.0/24 11 (content:"MECHANT";
pcre:"VRAI+MENT MECHANT"; msg:"PAQUET VRAIMENT MECHANT";)
```

Figure 2.1 Exemple d'une règle extraite de Snort

2.1.2 Les expressions régulières

Les expressions régulières sont des grammaires de type 3 dans la hiérarchie de Chomsky [18]. La première formalisation de ces dernières a été effectuée par Kleene [19]. Les expressions régulières permettent de décrire des successions d'événements. Elles sont notamment utilisées pour la recherche de texte et de motifs particuliers.

Une expression régulière se compose de deux types de caractères : les métacaractères et les caractères de base. Le tableau 2.1 donne un aperçu d'expressions et de leur métacaractère. Les caractères de base sont tous les autres caractères de la table ASCII. Le caractère \ est un caractère d'échappement, il permet soit de considérer un métacaractère comme un caractère de base ou bien de transformer un caractère de base comme un métacaractère. Ainsi, l'expression `\x40` valide `@`, en revanche l'expression `\*` valide `*`.

Tableau 2.1 Différents éléments des expressions régulières

Expression	Métacaractère	Correspondance
a.	.	Un *a* suivant de n'importe quel caractère autre que \n
*a**	*	0 ou plus *a*
a+	+	Au moins 1 *a*
a?	?	0 ou 1 *a*
A{i}	{i}	L'expression A apparaît i fois
A{i,}	{i,}	L'expression A apparaît au moins i fois
A{i,j}	{i,j}	L'expression A apparaît entre i et j fois
[^abcd]	[^ ...]	Tout caractère non listé (e, f, ...)
[abcd]	[...]	Un des caractères listés (a, b, c ou d)
(abc)	(...)	Groupe les éléments abc
A|*B*	|	Expression A ou Expression B

La figure 2.2 présente une expression régulière validant une fonction retournant un entier ou rien avec un nom composé d'au moins une lettre majuscule. Le nom de la fonction est suivi de parenthèses entre lesquelles il y a des paramètres séparés par des virgules. Les paramètres sont de type entier ou flottant et leur nom est composé d'au moins un chiffre ou caractère minuscule.

```
(int|void)[A-Z]+\( ((int|float)[a-z0-9]+) (,(int|float)[a-z0-9]+) \)
```

Figure 2.2 Exemple d'une expression régulière

2.1.3 Compilateurs

Dans cette section, quelques principes fondamentaux des compilateurs sont décrits. Le lecteur est invité à se référer au livre du dragon [20] pour de plus amples informations.

Un compilateur est un logiciel permettant de traduire un langage dans un autre, généralement un langage machine. Le flot de fonctionnement du compilateur est donné dans la figure 2.3 avec pour exemple la traduction de l'instruction VHDL `bascule <= (signal1 AND NOT signal2) OR (NOT signal1 AND signal2)` vers une instruction en langage C. Celui-ci se décompose en cinq étapes : l'analyse lexicale, l'analyse syntaxique, l'analyse sémantique, la génération et l'optimisation du code intermédiaire et la génération du code machine. Chacune des étapes partage la table de symboles qui peut être lue et écrite en fonction des besoins. Cette dernière contient notamment les différentes variables et fonctions créées.

L'étape d'analyse lexicale prend en entrée un flot de caractères qui est le code à traduire, et elle génère en sortie une suite de jetons. Les jetons permettent de standardiser la représentation en interne du compilateur. Lors de cette étape, la table de symboles est également remplie. La figure 2.3 montre la sortie de cette étape. Les variables `sorties`, `s1` et `s2` ont été respectivement assignés aux lignes 1, 2 et 3 de la table des symboles. Les opérateurs `<=`, `AND` et `OR` ont aussi été convertis en jetons.

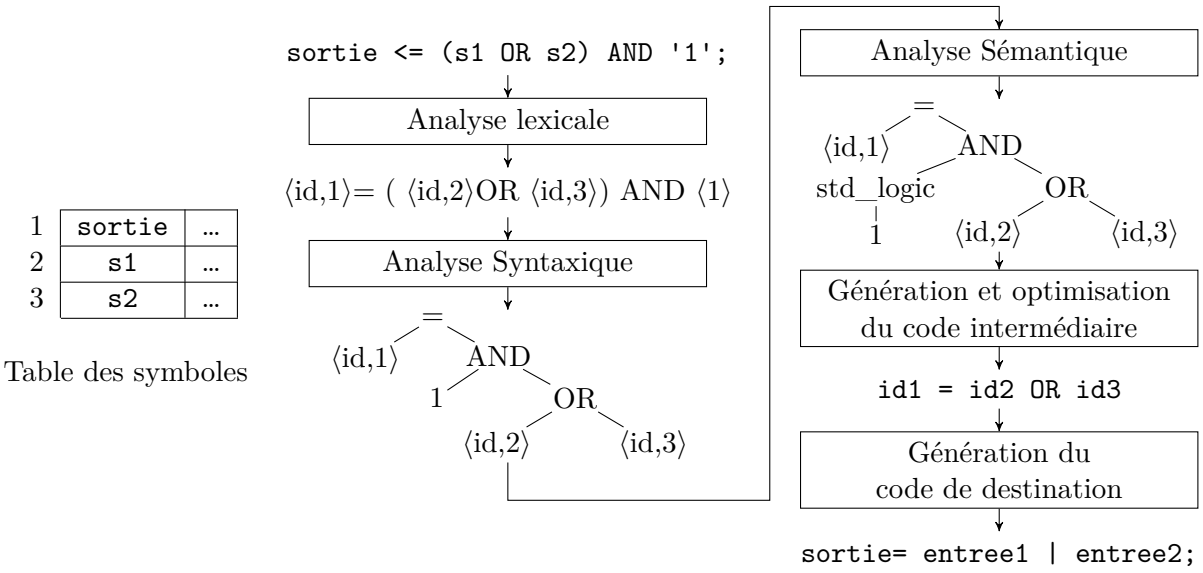


Figure 2.3 Les étapes d'un compilateur, exemple pour `sortie <= (s1 OR s2) AND '1';`

L'analyse syntaxique prend en entrée le flot de jetons et génère l'arbre syntaxique. L'arbre généré permet de représenter l'ordre dans lequel les opérations doivent être effectuées. Cet

arbre montre que l'on commence par effectuer `s1 OR s2`. Par la suite, on applique le ET logique `AND` au résultat obtenu avant. Enfin on assigne à `sortie` le résultat.

Lors de l'analyse sémantique, le compilateur vérifie la validité de sémantique par rapport à la définition du langage, par exemple qu'il y ait bien un `;` à la fin de l'instruction. Une partie importante de cette partie est la validation des différents types. Au moment de l'analyse sémantique, l'arbre syntaxique peut être développé pour déterminer le type de constante. Dans la figure 2.3, le `'1'` est défini comme un `std_logic`.

La génération du code intermédiaire et l'optimisation font une représentation intermédiaire propre au système. Dans l'exemple, une optimisation consiste à enlever le `'1' AND`. On obtient ainsi le code intermédiaire `id1 = id2 OR id3`.

Lors de la dernière étape, la génération du code de destination, ici du C, on crée le code de sortie. Ici on obtient le code C : `sortie = entree1 | entree2;`

2.1.4 Les automates

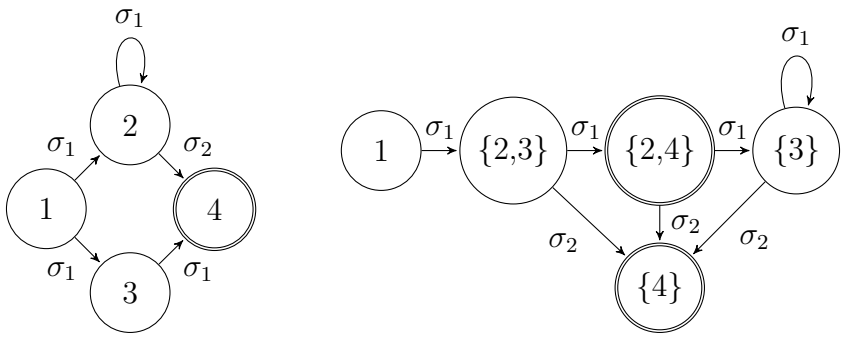
Les automates et les expressions régulières sont très liés étant donné que ces dernières ont été créées pour représenter les automates. Dans cette section, deux types d'automates sont présentés : les automates finis non déterministes — *Nondeterministic Finite Automaton* (NFA) et les automates finis déterministes (AFD). Pour plus d'information, le lecteur est invité à se référer aux livres de Salomaa [21] et Sakarovitch [22].

Les automates finis non déterministes : Un automate fini non déterministe — *Nondeterministic Finite Automaton* (NFA) $\mathcal{A}$ est un automate contenant un ensemble S d'états, avec une table d'entrée de symboles $\Sigma = \cup_{i \in [\![1,n]\!]} \sigma_i \cup \epsilon$ avec $n \geq 1$, un ensemble $I \subset S$ d'états initiaux, un ensemble $F \subset S$ d'états finaux et un ensemble T de transitions de $\mathcal{A}$. Dans cet automate, ϵ est une transition spontanée. Les NFA peuvent être représentés par un graphe tel que montré dans la figure 2.4a.

Les NFA peuvent avoir plusieurs états initiaux. Il peut y avoir plusieurs fonctions $f(s,a) \mid f \in T, s \in S$ et $a \in \Sigma$, il peut donc y avoir plusieurs états actifs en même temps. Par exemple, l'état 1 de la figure 2.4a possède deux transitions avec l'étiquette σ_1. L'automate présenté peut être représenté par l'expression $\sigma_1(\sigma_1|\sigma_1 * \sigma_2)$.

Les automates finis déterministes : Les AFD sont un cas particulier des NFA. Il s'agit d'automates n'ayant pas de transition ϵ et chaque fonction $f(s,a) \mid f \in T, s \in Seta \in \Sigma$ est unique, il n'y a donc qu'un état actif à la fois. Tout NFA peut être converti en un AFD, et la

taille du AFD résultant sera d'au maximum 2^n avec n le nombre d'états du NFA d'origine. Cette valeur provient du fait que chaque état du AFD permet de représenter les états actifs du NFA. Comme il y a n états dans le NFA, il y a au maximum 2^n états actifs en même temps. Un exemple de conversion du NFA de la figure 2.4a en AFD est donné dans la figure 2.4b. L'automate déterministe résultant possède un état de plus que l'automate non déterministe de départ. Chacun des états du AFD permet de représenter un ensemble d'états pouvant être actifs en même temps dans le NFA. Ainsi, lorsque l'état {2,3} du AFD est activé, cela signifie que le NFA aurait les états 2 et 3 d'actifs.



(a) NFA (b) AFD

Figure 2.4 Exemple de représentation d'automates

2.2 Travaux sur la recherche de texte

Cette section présente certaines solutions pour la recherche de listes de textes dans des flux de caractères.

2.2.1 L'algorithme Aho-Corasick

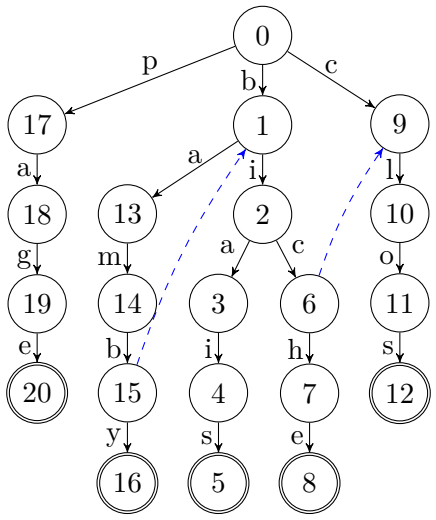
Une des premières solutions pour la recherche de texte a été proposée par Aho et Corasick en 1975 [23]. La solution proposée par les auteurs de l'article consiste à générer un AFD à partir d'un dictionnaire de mots et de le parcourir. L'algorithme se décompose en deux parties, la première consiste à créer l'automate, la seconde effectue la recherche.

Construction de l'automate : La machine complète résultante de l'automate est composée de trois fonctions, une pour avancer *"Goto"* ($g(S, A)$, avec S un état et A un caractère), une d'échec ($e(S)$) et une de sortie. La fonction de sortie permet de savoir lorsqu'une correspondance est trouvée. La fonction d'échec fait correspondre un état à d'autres états, cette

fonction permet de savoir à quel état se rendre lorsque la fonction *Goto* renvoie une erreur. Enfin, la fonction *Goto* permet d'avancer dans l'automate. Ces fonctions sont toutes développées lors de la création de l'arbre.

Un exemple d'automate pour le dictionnaire *biais, biche, bamby, clos, page* est présenté dans la figure 2.5. Dans cette figure, les fonctions d'échec renvoient par défaut à l'état 0. Lorsque cela n'est pas le cas, le trait bleu pointillé indique où cette fonction renvoie. Pour créer l'automate, l'état initial 0 est créé puis le premier mot du dictionnaire est ajouté, ici *biais*. Le mot suivant est ensuite ajouté en repartant de l'état initial. Ainsi, les états 1 et 2 de l'exemple traitant déjà *bi*, ils ne sont pas dupliqués.

Une fois l'automate construit, la fonction d'échec est réalisée. Tous les états de niveau 1, dans la figure 2.5a les états 1, 9 et 17, se voient assigner la valeur 0 pour le retour de leur fonction d'échec ainsi, $e(1) = e(9) = e(17) = 0$. Les états des autres niveaux sont ensuite traités niveau par niveau en donnant comme résultat à leur fonction d'échec le résultat de $f(e(S_{i-1}), A)$ avec S_{i-1} l'état précédent l'état actuel et A le caractère permettant d'arriver à S_i. Par exemple, pour l'état 6, son état précédent S_{i-1} est le 2 et le caractère de transition est c, on applique $f(e(2), c) = 9$, on a donc 9 comme retour de fonction d'échec. La table des fonctions d'échec résultante est présentée dans la figure 2.5b. Enfin, lors de la réalisation de la fonction d'échec, la fonction de sortie est aussi mise à jour au besoin. Cela arrive si, par exemple, un résultat d'échec amène à un état final.



(a) AFD

S	$e(S)$	S	$e(S)$
1	0	11	0
2	0	12	0
3	0	13	0
4	0	14	0
5	0	15	1
6	9	16	0
7	0	17	0
8	0	18	0
9	0	19	0
10	0	20	0

(b) Fonction d'échec

Figure 2.5 Exemple de machine de recherche générée par l'algorithme Aho-Corasick

Parcours de l'automate : Le parcours de l'automate permet de faire la recherche des mots du dictionnaire qu'il contient sur le flux d'entrée. Lors du traitement d'un caractère, on regarde si ce dernier est présent dans les transitions de l'état actuel. Si tel est le cas, alors on passe à l'état correspondant. Dans le cas contraire, on prend en compte la fonction d'échec et le caractère est testé de nouveau sur l'état. Enfin, si l'état actuel valide la fonction de sortie alors on indique une correspondance et on continue la recherche.

Par exemple, si l'on fait la recherche du flux de caractères *bambiche*, on commence à l'état 0, la fonction $g(0, b)$ active l'état 1. Le caractère a est ensuite testé, $g(1, a) = 13$, donc l'état 13 est activé, on continue le processus jusqu'à arriver à l'état 15 après avoir traité la chaîne *bamb*. À ce moment-ci, le résultat de $g(15, i)$ nous renvoie une erreur, la fonction d'échec est donc appelée. Lors de l'exécution de la fonction d'échec $e(15) = 1$, on retourne à l'état 1 et on effectue $g(1, i) = 6$, on va donc à l'état 6. On continue le processus jusqu'à arriver à l'état 8, après le traitement de *bambiche*, qui est un validant la fonction de sortie, on indique donc que le mot biche a été trouvé

2.2.2 Les implémentations de Aho-Corasick

La représentation des états d'un automate se fait à l'aide de structures mémoires appelées nœuds. La structure mémoire d'un nœud est donc l'élément déterminant sur la manière d'implémenter l'automate. La solution proposée par Aho et Corasick [23] pour représenter les nœuds consiste à utiliser un tableau contenant autant de colonnes que de symboles dans l'alphabet à traiter, et chacune des cases du tableau contient le pointeur permettant d'accéder au prochain nœud. Cette méthode permet de traiter chaque caractère d'entrée en un temps $O(1)$ mais requiert beaucoup de mémoire. Les travaux exposés dans cette section proposent des solutions permettant de réduire l'utilisation de mémoire tout en gardant le temps de traitement contant.

Une première méthode pour réduire la taille en mémoire d'un nœud utilise la compression matricielle. Cette méthode, utilisée par Tuck et al. [24], consiste à créer un nœud contenant un pointeur de base pour le nœud suivant, un pointeur pour le nœud d'échec et un vecteur de 256 bits dont chaque bit représente la validité d'un caractère. Lors du traitement d'un caractère, on regarde s'il est valide à l'aide du vecteur de caractères. Si ce dernier est valide, alors on calcule l'adresse du prochain nœud à l'aide d'un décalage par rapport au pointeur de base du prochain nœud. Si le caractère n'est pas présent, le nœud d'échec est utilisé. La solution proposée ici nécessite cependant une architecture matérielle dédiée pour le traitement des nœuds. En effet, le gain principal provient de l'utilisation du vecteur de bits, il faut donc rechercher à travers ce dernier.

Une seconde solution qui permet de réduire la taille de l'arbre dans son ensemble est la compression de chemin. Cette méthode proposée tout d'abord dans [24] améliorée par Bremler-Barr et al. [25], dérive de l'algorithme proposé par Morrison [26]. La compression de chemin s'applique pour des successions de nœuds qui n'ont qu'un prochain nœud. Pour ce faire, un nouveau type de nœud est créé et ce dernier contient un tableau de caractères ordonnés indiquant les caractères à valider. Le pointeur vers le nœud suivant indique l'adresse du dernier nœud. Par exemple, dans la figure 2.5a, les états 13, 14 et 15 seraient contenus dans un seul nœud et le pointeur du nœud suivant permettrait d'accéder au nœud représentant l'état 16.

Une autre méthode de compression de chemin est proposée par Bellekens et al. [27]. Les auteurs proposent de paralléliser la recherche de texte en utilisant un arbre Aho-corasick parallèle sans échec — *Parallel Failureless Aho-Corasick* (PFAC) proposé par Lin et al. [28] et implémenté sur un processeur graphique — *Graphics Processing Unit* (GPU). Dans le même temps, la recherche de préfixes est utilisées pour réduire l'empreinte mémoire de l'arbre. Cette méthode consiste à rechercher uniquement le début des préfixes. Si une correspondance pour un préfixe est trouvée alors la règle correspondante est cherchée. L'ajout de cette seconde méthode permet d'effectuer la recherche de texte au sein d'un GPU avec une empreinte mémoire plus faible que dans [28].

Une manière hybride est proposée par Lacroix et al. [29]. Les auteurs proposent deux types de nœuds. Le premier type contient un tableau contenant un couple *caractère,pointeur*, le second contient un tableau de pointeurs. Dans les deux types, le pointeur permet d'accéder au nœud suivant. Pour trouver le prochain nœud avec le premier type de nœud, le caractère courant est comparé à chacun des caractères dans le tableau. Dans le second type, le nœud suivant est déterminé par un accès direct au pointeur en utilisant le caractère courant comme index. Les auteurs proposent de faire un mélange entre les différents nœuds en fonction du nombre de prochains nœuds possibles. Cette solution peut nécessiter du matériel dédié dans les cas où le premier type de structure mémoire est utilisé avec beaucoup de caractères.

Tan et Sherwood [30] proposent une solution consistant à décomposer l'automate en sous-automates. La validation des résultats se fait alors avec un vecteur de bits. Dû à la taille importante que peut avoir le vecteur, ils proposent également de séparer le dictionnaire d'entrée en groupes et de générer autant d'automates que de groupes. Cette solution est également utilisée par Piyachon et Luo [31]. Ce concept est développé davantage dans la section 3.2.1. Un algorithme de groupement est également présenté dans la section 3.2.2.

2.3 Travaux sur la recherche d'expressions régulières

La section précédente présentait des travaux sur la recherche de texte. Dans cette section, différentes méthodes pour effectuer de la recherche d'expressions régulières sont exposées.

2.3.1 L'algorithme de Thompson

Thompson [32] propose une méthode pour faire la recherche d'expressions régulières. La méthode proposée par les auteurs se fait en deux étapes. Dans un premier temps, l'expression régulière est transformée en un NFA. Par la suite les auteurs présentent une manière de parcourir le NFA.

Comme un NFA peut avoir plusieurs états actifs en même temps, le parcours se fait en utilisant une liste *l1* des états actifs. Lorsqu'un caractère est traité, la liste des états actuels est parcourue. Pour chacun des états, on regarde si une transition correspond à ce dernier. Chacun des états suivants est ajouté à la prochaine liste *l2* d'état actif. Une fois la liste *l1* parcourue, on la remplace par *l2*.

2.3.2 Des implémentations de recherche d'expressions régulières

Comme le parcours des NFA ne se fait pas dans un temps déterministe, une approche pour rendre ce dernier déterministe réside dans l'implémentation du NFA sur des architectures parallèles.

Floyd et Ullman [33] développent un modèle de circuit permettant d'intégrer un NFA de manière simplifiée. La représentation des états du NFA se fait en assignant un bit par état. Cet article montre que dans le cas où le coût en espace des différents éléments logiques du circuit est constant et que les fils sont de taille fixe, l'aire nécessaire à l'intégration du NFA est proportionnelle à la taille de celui-ci.

Sidhu et Prasanna [34] ainsi que Mitra et al. [35] proposent d'utiliser l'algorithme de Thompson pour compiler une expression régulière pour un FPGA. Comme pour [33], l'objectif est de simplifier l'intégration de ce dernier. Chacun des états de l'automate est représenté par une bascule D, il y a donc autant de bascules que d'états. Cette représentation permet de traiter chacun des états en parallèle. La création des éléments du circuit se fait lors de la compilation. L'expression régulière est lue caractère par caractère. En considérant que le placement et routage s'effectuent en un temps constant par caractères lus de l'expression régulière, le temps d'implémentation est linéaire. Une fois implémentée, la recherche d'une expression se fait en temps linéaire.

Une architecture pour FPGA permettant la recherche d'expressions régulières sur plusieurs flots est proposée par Qu et al. [36]. Le système proposé est multi-contexte et s'intéresse à une solution permettant de stocker ces derniers dans une mémoire externe au FPGA. Les flots à comparer et les expression régulières sont comparées avec l'aide de plusieurs pipelines. Lors de la recherche d'une expression régulière, le chargement d'un flot est effectué pendant que la recherche d'expression dans un autre flot est faite. Dans le même temps, les expressions régulière sont aussi chargé sur les pipelines disponibles. La solution proposée permet au final de chercher toutes les expressions régulières dans tous les flots de donnée sans être impactée par la latence de la mémoire externe. Cette caractéristique permet de supporter un grand nombre d'expressions régulières sur un FPGA.

Une solution utilisant des processeurs *Sub-RISC* est proposée par Mihal et al. [37]. L'idée est d'implémenter plusieurs processeurs spécialisés dans le traitement des expressions régulières pour faire la recherche de celles-ci. Les processeurs *Sub-RISC* sont reprogrammables, mais moins polyvalents que des processeurs RISC [38]. L'objectif de ce concept est de proposer un intermédiaire entre le RTL et le logiciel. Pour effectuer la construction du système, le même procédé que [34] est utilisé, mais plus d'éléments du langage sont supportés. Le système permet de rechercher 385 expressions régulières. Pour cela, 55 éléments programmables sont implémentés en parallèle avec une mémoire d'instructions de 256 éléments. Le nombre de cycles pour traiter un caractère étant égal à la profondeur de la mémoire d'instructions, 256 cycles sont nécessaires. Cependant, une augmentation du nombre d'éléments programmables peut permettre une réduction du nombre de cycles nécessaires par caractère.

Les GPU étant des architectures massivement parallèles, Vasiliadis et al. [39] proposent de les utiliser pour faire de la recherche d'expression régulières dans le cas des IDS en y implémentant des AFD. La parallélisation du système s'effectue au niveau du traitement des paquets. Ainsi, chaque fils d'exécution sur le GPU permet de comparer un paquet. Pour chaque paquet à comparer, il a été déterminé en amont l'expression régulière à chercher. Afin d'éviter une consommation de mémoire trop importante suite à l'augmentation importante d'états dû à la conversion du NFA en AFD, une valeur de seuil est déterminée. Si le AFD d'une expression régulière dépasse cette valeur, alors la recherche de cette expression est faite sur le processeur générique.

Les GPU sont également utilisés par Wang et al. [40] pour effectuer la recherche d'expressions régulières. Les auteurs utilisent deux méthodes pour améliorer les latences d'accès à la mémoire. La première méthode utilisée consiste à améliorer le transfert de données entre l'hôte et le GPU par un mécanisme asynchrone. Ce mécanisme permet d'effectuer le transfert de données sans interrompre les fils d'exécution. La seconde méthode adapte le stockage

des données et leur accès à la structure spécifique de la mémoire du GPU. Ceci se fait en chargeant les paquets dans la mémoire partagée depuis la mémoire globale du GPU. Cette mémoire partagée est également structurée de manière à éviter les conflits d'accès.

Lunteren et Guanella [41] proposent d'utiliser un nouveau type de machine à état, les *B-FSM* [42], qui utilisent une fonction de hachage avec un nombre fixe de collisions. Le système contient une table de règles de transitions. Une règle de transition détermine en fonction de l'état courant et de l'entrée quel est l'état suivant. Chaque règle a une priorité qui lui est donnée afin de gérer les conflits. La fonction de hachage permet de déterminer quelles règles de transition pourraient être valides dans un état donné. Comme le nombre de collisions maximales est connu à l'avance, la comparaison des différentes règles de transitions peut se faire en un seul cycle. Enfin, les gros AFD sont gérés en séparant ces derniers entre plusieurs *B-FSM* [43].

Divyasree et al. [44] présentent une architecture reconfigurable pour la recherche d'expressions régulières. L'architecture proposée est une matrice de blocs génériques dont chacun des blocs supporte toute syntaxe d'expression régulière à un caractère. Pour que les blocs soient reconfigurables, les auteurs proposent d'utiliser des registres à décalages de 32 bits inhérents aux tables de correspondance — *Look-Up Table* (LUT) des FPGA. La reconfiguration d'une LUT nécessite ainsi 32 cycles d'horloge. Par ailleurs, la largeur du bus de reconfiguration étant de 64 bits, il est possible de reconfigurer 64 LUT en même temps. Au final, il est possible de reconfigurer 8 blocs génériques tous les 32 cycles d'horloge. La solution proposée est donc un système rapidement reconfigurable ce qui permet d'implémenter une expression régulière à la fois et de la changer en fonction du besoin. Cette solution implique toutefois l'utilisation de 6 fois plus de ressources qu'une architecture dédié à une seule expression régulière.

2.4 Travaux sur les architectures intermédiaires

Les architectures intermédiaires, traduction de *intermediate fabric*, aussi appelées *overlay*, sont un concept d'architecture reconfigurable, mais implémentée sur un FPGA. Cette section s'intéresse à différents travaux réalisés dans ce domaine.

Bergmann et al. [45] ont proposé ce concept et l'ont utilisé en implémentant un filtre à réponse impulsionnelle finie. La solution proposée consiste à avoir deux niveaux de configuration. Un pour reconfigurer l'architecture en quelques millisecondes, et le second niveau pour reconfigurer l'architecture à gros grain à des rythmes de l'ordre du cycle d'horloge. Le premier niveau permet ainsi d'adapter l'architecture à de nouvelles applications. Les auteurs proposent d'implémenter plusieurs blocs d'architecture à gros grains dans une matrice programmable. Le

système complet peut donc être vu comme une matrice programmable multi-instructions, multi-données. Dans le cas de l'implémentation du filtre, pour une utilisation des ressources stable indépendamment de la taille du filtre, le système résultant est plus performant qu'une implémentation sur un processeur, mais moins performant qu'une architecture dédiée. Ce résultat provient de la capacité des éléments reprogrammables à être reconfigurés sur des temps très courts, ainsi il est possible d'utiliser une même ressource dans plusieurs étapes du calcul. Ceci n'est pas possible sur les FPGA actuels, car le temps de reconfiguration de ce dernier, même partiel, est trop long par rapport au temps du calcul.

Coole et Stitt [46] étudient le gain et le surcoût en ressources d'utiliser une architecture intermédiaire. Les résultats montrent un surcoût moyen de 18% qui a été réduit à 10% dans le cas de circuits plus spécialisés. La spécialisation présentée consiste principalement à limiter les possibilités de routage. Cette spécialisation requiert une réduction de la flexibilité du circuit de seulement 9% selon les auteurs. Il y a également en moyenne 18% de diminution de la fréquence d'horloge par rapport à une implémentation directe sur FPGA. Toutefois, pour certaines applications, l'architecture permet de meilleures fréquences d'horloge. Enfin, ces différents désagréments sont à mettre en perspective du gain moyen en temps pour le placement et routage qui est divisé par 554 en moyenne.

Lysecky et al. [47] proposent un FPGA virtuel à implémenter sur un FPGA physique. L'objectif est de permettre une meilleure portabilité d'architecture matérielle. Pour cela, une architecture est implémentée pour le FPGA virtuel puis cette dernière peut être mise sur tout FPGA implémentant le FPGA virtuel. Il suffit donc d'uniquement implémenter le FPGA virtuel pour des FPGA physiques. Ceci permet dans le cas de système déployé à large échelle de propager des mises à jour sans avoir à ré-implémenter la nouvelle architecture pour tous les FPGA de destination. Brant et Lemieux [48] proposent une architecture appelée ZUMA, permettant de diviser par plus de 2x le surcoût d'implémentation par rapport à l'implémentation proposée par [47]. La solution proposée nécessite cependant des ressources particulières sur le FPGA de destination contrairement au travail de Lysecky et al.

2.5 Retour sur la problématique

Un sommaire des travaux exposés dans les sections 2.2, 2.3 et 2.4 est présenté dans le tableau 2.2. Le tableau se décompose en trois parties, identifiées par la première colonne, la recherche de texte, la recherche d'expressions régulières et les architectures intermédiaires. La seconde colonne identifie les auteurs, l'année et référence le travail. La troisième colonne présente l'approche du travail. La quatrième colonne identifie la technologie sur laquelle le travail a été implémenté. La cinquième colonne donne des compléments d'information.

Dans les chapitres suivants, les contributions de ce mémoire sont présentées. Les travaux du chapitre 3 concernent la recherche de texte et s'appuient principalement sur les travaux de Piyachon et Luo [31]. Les travaux du chapitre 4 concernent la recherche d'expressions régulières et s'appuient sur les travaux de Thompson [32] et de Sidhu et Prasanna [34]. Enfin les travaux du chapitre 5 concernent le concept des architectures intermédiaires et s'appuient sur le travail de Sidhu et Prasanna [34].

Tableau 2.2 Résumé des travaux antérieurs

	Référence	Approche	Technologie	Notes
Recherche de texte	Aho et Corasick 1975 [23]	AFD	CPU	Génération de l'AFD
	Tuck et al. 2004 [24]	Compression des nœuds	ASIC/CPU	Correspondance de bits
	Bremler-Barr et al. 2011 [25]	Compression de chemins	CPU	Regroupement des états avec une seule transition
	Bellekens et al. 2014 [27]	DFA	GPU	PFAC et correspondance de préfixes
	Lacroix et al. 2016 [29]	Compression des nœuds	CPU	Plusieurs types de nœuds
	Tan et Sherwood 2005 [30]	Bit-split	ASIC	Séparation de l'automate et des règles
	Piyachon et Luo 2006 [31]	Bit-Byte-Split	Processeur réseau	Groupement des règles
Recherche d'expressions régulières	Thompson 1968 [32]	NFA	CPU	Génération d'un NFA
	Floyd et Ullman 1982 [33]	NFA	ASIC	Modèle simplifié d'implémentation de NFA
	Sidhu et Prasanna 2001 [34]	NFA	FPGA	Compilation du jeu d'expressions sur un FPGA
	Qu et al. 2012 [36]	NFA	FPGA	Contexte dans une mémoire externe
	Mitra et al. 2007 [35]	NFA	FPGA	Supporte le langage PCRE
	Mihal et al. 2006 [37]	NFA sub-risc	Sub-risc	NFA sur des processeurs sub-risc
	Vasiliadis et al. 2009 [39]	AFD	GPU	Parallélisation du traitement des paquets
	Wang et al. 2011 [40]	DFA	GPU	Adaptation pour la structure mémoire du GPU
	Lunteren et al. 2012 [43]	B-FSM	Processeur réseau	Processeur réseau d'IBM
	Divyasree et al. 2008 [44]	NFA	FPGA	Architecture reconfigurable
Architectures intermédiaires	Bergmann et al. 2013 [45]	Gros grains	FPGA	Reconfiguration à deux niveaux
	Coole et Stitt 2010 [46]	Gros grains	FPGA	Comparaison d'applications
	Lysecky et al. 2007 [47]	Grains fins	FPGA	FPGA sur un FPGA
	Brant et Lemieux 2012 [48]	Grains fins	FPGA	FPGA sur un FPGA

CHAPITRE 3 UNE ARCHITECTURE POUR LA RECHERCHE DE CHAÎNES DE CARACTÈRES

Les règles des IDS sont représentées, en partie, avec des chaînes de caractères. Avec la prochaine génération de réseau, ces systèmes doivent être capables de faire la recherche de règles avec une faible latence et un grand débit. Dans ce contexte, un des objectifs est d'accélérer la recherche de texte. Dans ce chapitre, nous présentons une architecture permettant de faire la recherche de chaînes de caractères. Une première section présente les contraintes majeures des IDS. Puis, la seconde section explique certains concepts essentiels. La troisième section détaille l'architecture réalisée. Par la suite, la quatrième section donne les résultats d'implémentation. Finalement, dans la cinquième section une analyse des différents points concernant l'architecture est présentée.

3.1 Deux contraintes majeures pour les IDS

Les IDS doivent satisfaire deux contraintes principales : ils doivent être déterministes dans le temps de traitement et ils doivent opérer avec un grand débit. La première contrainte correspond à une limite sur le temps de traitement maximal d'un flux de données. Le système doit être à même de produire des résultats dans un temps raisonnable connu et il ne doit pas être vulnérable aux attaque de déni de service — *Denial of Service* (DoS). Une attaque de DoS consiste à envoyer à un système une requête qui l'occupe suffisamment afin de le rendre indisponible pour les autres requêtes. La deuxième contrainte relève des besoins opérationnels du système et ultimement indique combien de systèmes parallèles doivent être activés afin de répondre aux besoins de l'opérateur du réseau. De plus, comme beaucoup de données sont traitées, le traitement doit être rapide.

Afin d'obtenir un temps de traitement borné, les IDS utilisent majoritairement des algorithmes dérivés de l'algorithme de Aho-Corasick [23]. Tel qu'expliqué dans la section 2.2, cet algorithme génère un AFD à partir d'un dictionnaire, et l'utilisation du AFD résulte en un temps de traitement déterministe. L'implémentation logicielle de ces automates consiste à utiliser un arbre préfixé. Un inconvénient de cette approche est sa grande consommation de mémoire. Il est donc nécessaire de trouver des façons de réduire les besoins en mémoire, et c'est une des caractéristiques de l'architecture utilisée dans ce chapitre.

3.2 Le concept utilisé et l'algorithme de groupement

Cette section présente deux concepts utilisés pour réaliser le système de recherche de chaîne de caractère. La première section s'intéresse au concept de la séparation d'octets en bits *bit-byte split*. La seconde section présente l'algorithme de groupement utilisé.

3.2.1 La séparation d'octets en bits *bit-byte split*

Piyachon et Luo proposent la répartition des octets en bits afin de réduire l'empreinte mémoire d'un AFD [31]. La méthode proposée consiste à utiliser le grand nombre d'éléments programmables (EP) disponibles dans les processeurs réseau (PR). Pour cela, la méthode consiste à compiler les règles d'entrées en prenant deux paramètres, B le nombre d'octets à considérer pour une transition, et b le nombre de bits par lesquels un octet est divisé.

Augmenter la valeur de B réduit le nombre d'états N de l'automate. Toutefois, il faut parcourir B fois l'automate afin de s'assurer de tester chacun des caractères de départ possibles. Cela est, en pratique, réalisé en implémentant plusieurs fois l'automate. Cette approche est intéressante lorsque les chaînes de caractères sont très peu corrélées entre elles. Effectivement, s'il n'y a aucune intersection entre les n chaînes de caractères d'une liste, on obtient $N = \sum_{i=1}^{n} N_i$ avec N_i le nombre d'états de l'automate généré pour la i-ème chaîne de caractères de la liste. La figure 3.1 montre les automates résultants pour les chaînes *"agricole"* et *"chasseur"* dans les cas ou $B = 1$ et $B = 4$.

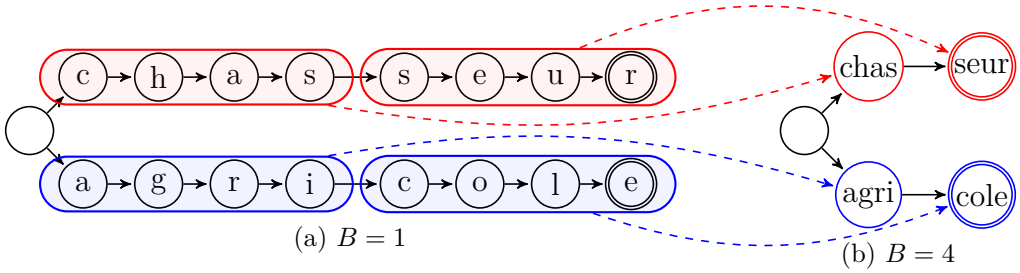


Figure 3.1 Automate résultant de *"agricole"*, *"chasseur"*, pour (a) $B = 1$ et (b) $B = 4$

Le découpage d'un AFD en plusieurs machines à états finis — *Finite State Machine* (FSM) permet de diminuer la redondance des comparaisons. En effet, supposons un AFD avec un alphabet de transition sur 8 bits, et prenons un état Σ ; celui-ci possède un maximum de 256 transitions. Supposons maintenant que Σ possède uniquement deux transitions t_1 et t_2 n'ayant qu'un bit de différence. Le stockage des transitions dans ce cas-ci nécessite $2 \times 8 = 16$ bits. Si maintenant l'automate est décomposé en huit sous-automates $\sigma_1 - \sigma_8$ avec un alphabet de transition de 1 bit, alors les transitions t_1 et t_2 sont réparties entre les différents

automates. Comme seulement 1 bit différencie les transitions, sept des huit automates auront une unique transition pour stocker t_1 et t_2. Le huitième automate possède deux transitions pour enregistrer la variation d'un bit. Il sera donc nécessaire d'enregistrer un total de 9 transitions de 1 bit soit 9 bits pour enregistrer la table de transitions. Ce gain de mémoire n'est efficace que dans le cas où un état possède moins de transitions que le nombre d'éléments de l'alphabet.

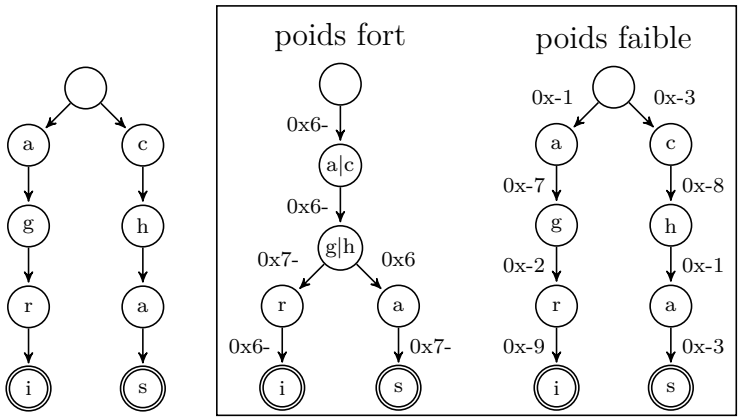
La figure 3.2 montre pour l'ensemble $\{$*"agri", "chas"*$\}$, les automates dans les cas où $b = 8$ et $b = 4$. L'automate de départ possède 8 transitions alors que les deux automates résultants ont un total de 14 transitions. Cependant, chacun des états dans le deuxième automate possède un alphabet de transition sur 4 bits alors que celui d'origine est sur 8 bits. La mémoire nécessaire pour représenter les transitions passe de $8 \times 8 = 64$ bits à $14 \times 4 = 56$ bits. Le gain total dépend de la densité des transitions par état et donc de la corrélation entre les règles.

3.2.2 Algorithme de réduction de la mémoire dans bit-byte split

Vakili et al., proposent un algorithme permettant de réduire l'utilisation de la mémoire pour la recherche de chaînes de caractères [49]. Pour cela, les auteurs proposent de grouper les règles selon des critères prenant en compte le concept décrit dans la section 3.2.1. La méthode proposée consiste à assigner une règle dans un groupe en fonction du nombre d'états qu'elle a en commun avec celui-ci.

Pour cela, l'algorithme prend en entrée le nombre de FSM divisées en bits - *Bit-Split FSM* (BS-FSM) disponibles ainsi que l'ensemble C des chaînes de caractères. L'algorithme s'exécute en deux phases. La première initialise chacun des groupes avec une des chaînes de caractères de C. Chacune des chaînes est sélectionnée en prenant la chaîne la moins corrélée avec les autres. Dans la seconde phase, les chaînes restantes de C sont assignées aux différents groupes en trois étapes. Dans la première étape, une chaîne de caractère C_j est sélectionnée telle que $\forall C_i \in C, m1_j - m2_j \geq m1_i - m2_i$ avec $m1_i$ la valeur de corrélation du groupe le plus proche avec C_i et $m2_i$ la valeur de corrélation de C_i avec son deuxième groupe le plus proche. La seconde étape assigne C_j au groupe n avec lequel la chaîne à la plus grande corrélation. La troisième étape effectue la mise à jour de la corrélation du groupe n.

La corrélation entre deux chaînes est calculée en fonction du nombre d'états nécessaires pour les représenter. Cette méthode permet une réduction de la consommation de mémoire en groupant les règles en fonction de leurs similarités et de l'architecture de destination. Toutefois, l'utilisation de cet algorithme nécessite de comparer simultanément tous les groupes lors de la comparaison d'un flux de caractères.



(a) $b = 8$ (b) $b = 4$

Caractère	code Hex
a	0x61
c	0x63
g	0x67
i	0x69
r	0x72
s	0x73

(c) code hexadécimal

Figure 3.2 Transformation d'un automate pour (a) $b = 8$ vers (b) $b = 4$ résultant de *{"agri", "chas"}* et (c) code hexadécimal de certains caractères

3.3 L'architecture réalisée

Dans la section précédente les éléments de base ont été présentés. Cette section se focalise sur l'architecture réalisée plus en détail. Dans un premier temps, les différentes décisions d'implémentation seront présentées. Par la suite, le fonctionnement de l'architecture sera détaillé.

3.3.1 L'architecture du système

La figure 3.3 présente le design réalisé. Cette architecture se base sur l'algorithme présenté dans la section 3.2.2 ainsi que sur le concept des octets divisés en bits présenté dans la section 3.2.1. L'architecture se décompose en deux éléments principaux, les EP et les BS-FSM. Les BS-FSM contiennent des unités de comparaison de motifs — *Pattern Matching Units* (PMU) qui effectuent le parcours des automates. Chaque EP contrôle une BS-FSM et est associé à un groupe dans l'algorithme de groupement. Toutes les BS-FSM ont en entrée B caractères et un signal provenant de chaque EP et indiquant si un résultat a été trouvé. Chaque BS-FSM retourne un signal indiquant si un résultat a été trouvé. Les EP sont connectés au système hôte, à un signal d'activation et à un bus connecté à chaque PMU de sa BS-FSM contenant les caractères ainsi que des signaux de contrôles.

La figure 3.4 présente une vue détaillée de l'implémentation d'une BS-FSM traitant des mots de quatre bits et possédant deux paires de PMU de un bit par PMU. Pour permettre une meilleure utilisation des mémoires, une mémoire statique à accès direct à double accès — *Dual*

Port Static Random Access Memory (DP-SRAM) est utilisée pour effectuer l'implémentation d'une PMU. Cela permet d'améliorer l'utilisation de la mémoire en mettant en commun plusieurs éléments. Finalement, comme il y a une redondance entre les résultats des différentes PMU, le vecteur de résultats est enregistré dans une mémoire. Une fois un résultat sélectionné, un *ET* logique est effectué entre les différents bits des vecteurs de résultats. Le signal *Result Rdy* indique la validité du résultat.

3.3.2 Fonctionnement du système

La section précédente présentait l'architecture du système. Cette section s'intéresse au fonctionnement de l'architecture. En premier lieu, une présentation générale du processus d'utilisation sera développée. Puis, le fonctionnement des BS-FSM sera expliqué. Finalement, la récupération du résultat sera présentée.

Processus global : Dans un premier temps, le processus présenté dans la figure 3.5 est exécuté. Lors de la réception d'une base de règles, leurs contenus sont extraits puis analysés. Une fois l'analyse effectuée, les paramètres d'implémentations (B, b) sont définis. Ces premières étapes permettent de configurer une architecture dont l'implémentation sera affinée en regard des règles reçues. Lors de la génération des paramètres d'implémentation, les futures évolutions possibles sont prises en comptes afin d'éviter de devoir refaire l'implémentation après chaque mise à jour. Par la suite, le circuit est implémenté et les BS-FSM sont configurées.

Une fois le système opérationnel, les EP traitent le flux d'entrée et l'envoient aux BS-FSM. Finalement, les résultats des différentes BS-FSM sont comparés par leur EP respectif. Dès qu'une correspondance est détectée, un signal est envoyé à tous les EP. Ce signal remet les BS-FSM dans leur état initial en attendant une future requête de recherche. L'identifiant de la règle est envoyé au système hôte.

Fonctionnement des BS-FSM : Les BS-FSM fonctionnent selon deux modes, un premier qui permet leur configuration et un second pour effectuer les recherches. Dans le premier mode, les mémoires des différentes PMU sont chargées. Ce mode est activé une première fois lors de l'initialisation du système. Le système est ensuite prêt à être utilisé. Pour cela, les EP traitent les données entrantes pour les envoyer aux BS-FSM. Lorsque la modification de la mémoire d'une BS-FSM est nécessaire, celle-ci s'arrête puis la mise à jour du contenu des mémoires est effectuée. Le contenu des mémoires est déterminé en amont à la suite de l'exécution de l'algorithme présenté dans la section 3.2.2.

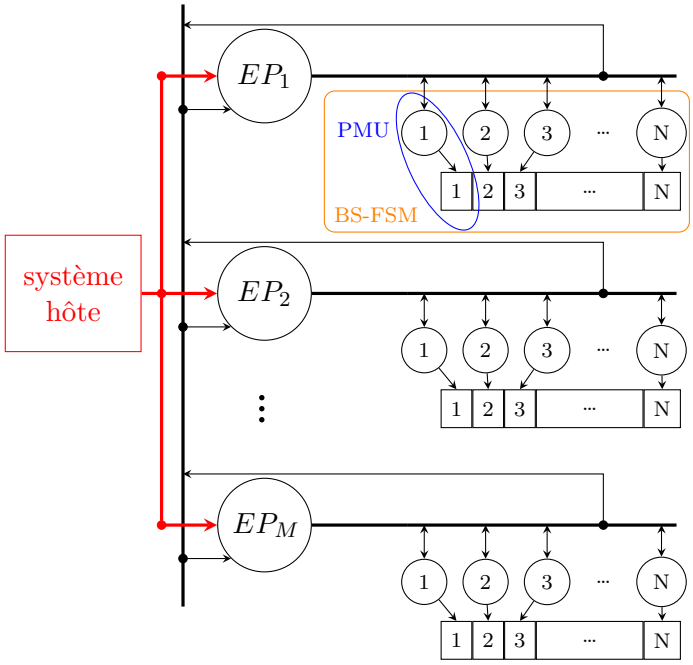


Figure 3.3 Architecture générale de l'implémentation de M BS-FSM

Récupération du résultat : Chacune des PMU est composée de deux mémoires, une de transitions et une de résultats. La mémoire des transitions contient un pointeur vers la mémoire des résultats qui entrepose le vecteur de résultats. Le vecteur de résultats est un mappage binaire, dans lequel la position du bit correspond à un numéro de règle. Ainsi, lorsqu'un bit est à 1, une correspondance pourrait avoir lieu pour la règle désignée. Celle-ci doit être comparée aux autres FSM tel qu'expliqué dans la section 3.2.1. Un ET logique est effectué entre tous les vecteurs de résultat des PMU. La validation du résultat s'effectue par un OU logique et le numéro de la règle correspondante est récupéré à l'aide d'un décodeur puis est transmis au système hôte.

3.4 Résultats d'implémentation

Cette section présente les résultats d'implémentation de l'architecture. L'implémentation a été réalisée avec les paramètres $B = 1$, $b = 1$ pour le nombre d'octets traités par BS-FSM et le nombre de bits traités par PMU. Ces paramètres ont été déterminés en amont par l'exécution de l'algorithme de groupement sur environ 3000 règles de Snort v2.9. Par la suite, l'implémentation a été réalisée pour les cas ou il y a 16, 32 ou 64 BS-FSM (i.e : groupes) considérés. Les trois versions ont été décrites avec le langage VHDL, en utilisant le logiciel *Vivado 2014.4* et implémentées sur un FPGA Xilinx®*XC7Z100*. La suite de cette section

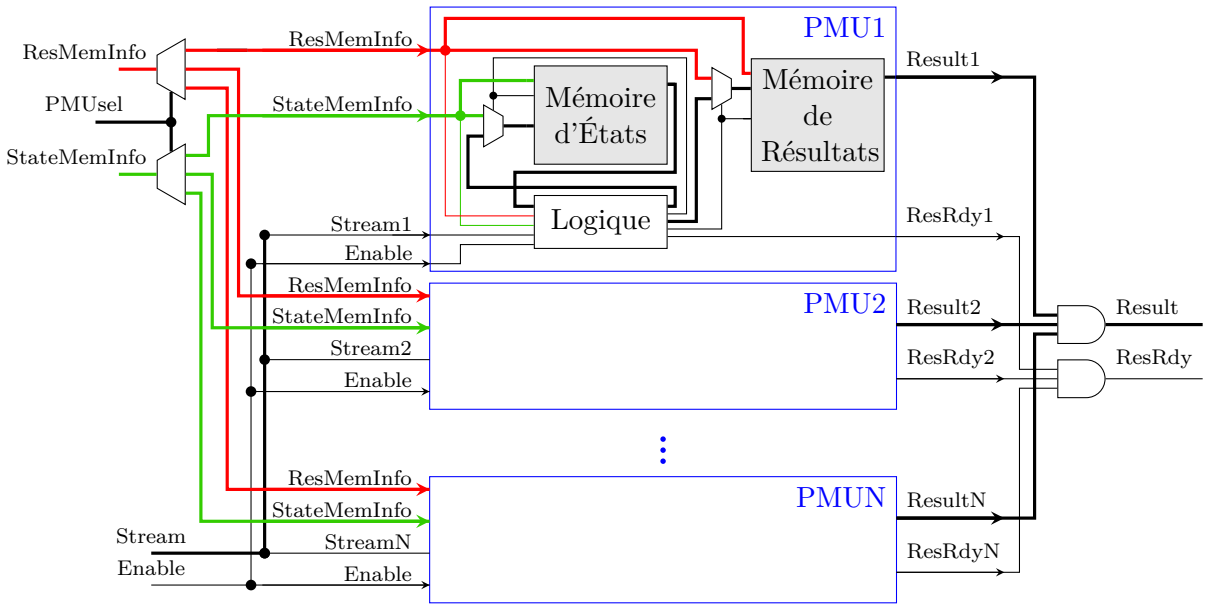


Figure 3.4 Schéma de l'implémentation d'une BS-FSM avec N PMU et $b = 1$ bit par PMU.

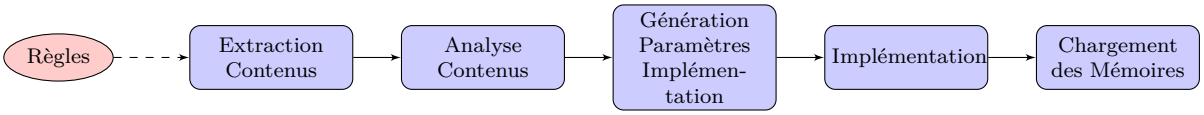


Figure 3.5 Processus pour l'utilisation d'une BS-FSM.

présente les résultats résumés dans le Tableau 3.1. Les ressources consommées sont indiquées par BS-FSM, la colonne *Total* est le produit de la ressource avec le nombre de BS-FSM.

Les résultats montrent une diminution de toutes les ressources consommées par BS-FSM. Le nombre de LUT passe de 537, lorsqu'il y a 16 groupes, à 226 pour 64 groupes soit une division par presque 2. Le nombre de Blocs Mémoires (BM) est lui diminué d'un ordre de grandeur. Comme il y a un minimum de deux BM par BS-FSM, cette valeur est minimum. Le nombre de registres est divisé par plus de 3. Finalement, la période d'horloge reste stable à 2 ns.

En plus de la diminution des ressources consommées par BS-FSM, on remarque également une division par deux du nombre de BM nécessaires pour implémenter 64 BS-FSM par rapport au total nécessaire pour l'implémentation de 16 BS-FSM. On remarque également que malgré une augmentation strictement croissante du nombre total de LUT nécessaires, le nombre total de registres ne suit pas cette tendance ; leur total diminue lorsque l'on passe de 32 à 64 BS-FSM.

La diminution du nombre de mémoires nécessaires par BS-FSM s'explique par la diminution du nombre de règles par groupes. En effet, dans le cas de 16 groupes, il y a au mieux

$\lceil 3000 \div 16 = 187 \rceil$ règles par groupe, alors que dans le cas de 64 groupes, ce nombre descend à 47 règles. Étant donné que le vecteur de résultats dans un groupe est lié au nombre de règles, et que les BM sont de taille fixe, la réduction de la taille du vecteur de résultats permet d'utiliser moins de BM. Étant donné qu'une BS-FSM a besoin d'au moins deux BM, l'augmentation du nombre de groupes passé 64 n'aurait pas d'influence sur cette métrique ; c'est pour cela que les implémentations ont été faites jusqu'à 64 groupes seulement.

Tableau 3.1 Résultats d'implémentation des BS-FSM en fonction du nombre de groupes.

	LUT		Bascules D		BM		
Nb BS-FSM	par BS-FSM	Total	par BS-FSM	Total	par BS-FSM	Total	Période
16	537	8,6k	957	15,3k	22	352	2,038 ns
32	321	10,3k	533	17,1k	12	384	2,018 ns
64	226	14,5k	265	16,7k	2	128	2,045 ns

3.5 Discussion

L'architecture décrite dans cette section permet une implémentation efficace de l'algorithme proposé par Vakili et al. [49]. La faible empreinte mémoire des automates résultant de l'algorithme de groupement et le haut niveau de parallélisme nécessaire pour le parcours des automates rendent l'usage d'une plateforme FPGA particulièrement intéressante. L'utilisation d'un FPGA laisse la possibilité à de futures évolutions pour s'adapter aux tables de règles. Comme le système a une faible empreinte mémoire, il est possible de n'utiliser que des mémoires intégrées et donc de traiter un caractère par cycle. Finalement, la plateforme FPGA laisse la possibilité d'une mise en service rapide et donne donc une souplesse dans son utilisation en comparaison d'un circuit intégré développé pour un client - *Application Specific Integrated Circuit* (ASIC).

L'architecture proposée a deux contraintes principales. La première est liée aux mises à jour, la seconde concerne l'encodage des résultats.

Dans les systèmes réseau, les mises à jour sont un élément important. Il faut que celles-ci se fassent rapidement et sans interrompre le traitement. Le système de recherche de chaînes de caractères proposé ne permet pas de rencontrer ces deux contraintes pour effectuer des mises à jour. Effectivement, étant donné que les résultats de toutes les BS-FSM sont nécessaires et que les règles sont réparties indépendamment de leurs caractéristiques, il n'est pas possible de mettre à jour chacune des BS-FSM de manière indépendante. Il est possible que, lors de l'exécution de l'algorithme de groupement, certaines règles soient déplacées d'un groupe vers

un autre, ou bien que la modification des mémoires d'états puisse induire un état indéterminé lors du fonctionnement. Plusieurs solutions existent pour pallier à ce problème. Une solution consiste à utiliser deux systèmes simultanément et de les mettre à jour de manière séquentielle. Une seconde solution consiste à doubler les mémoires dans les BS-FSM. Lors d'une mise à jour, la deuxième mémoire est remplie ; une fois le système entièrement mis à jour cette deuxième mémoire est utilisée à la place de la première. Une troisième solution consiste à utiliser les zones libres des mémoires des différentes BS-FSM pour faire une mise à jour des machines à états de manière incrémentale. Cette méthode nécessite une opération supplémentaire en amont pour déterminer les zones mémoires libres et les ordres de mises à jour. Elle nécessite également un espace suffisant pour l'ajout des nouvelles règles et des nouveaux états.

Dans le système proposé, l'encodage du résultat en utilisant un tableau de bits implique un surcoût important, notamment sur les capacités d'évolution du système. Si le nombre de règles dans un groupe dépasse la taille du vecteur de résultats alors il faut générer une nouvelle architecture pour supporter les règles supplémentaires. La taille du vecteur de résultats a également un impact direct sur le nombre d'éléments mémoire nécessaire. Ceux-ci ayant un bus de données avec une largeur maximale, l'ajout d'un résultat peut mener à l'utilisation d'un bloc mémoire complet. Une solution pour ce problème consisterait à utiliser les bloc de Logique Programmable — *Configurable Logic Block* (CLB) mémoires, cela pourrait cependant avoir un impact sur la latence.

CHAPITRE 4 COMPILATEUR D'EXPRESSIONS RÉGULIÈRES VERS UN FPGA

En plus des chaînes de caractères, les règles des IDS contiennent des expressions régulières. Ce chapitre s'intéresse à la réalisation d'un compilateur d'expressions régulières (regex) vers le VHDL. La première section de ce chapitre traite des contraintes liées à la recherche de expressions régulières (regex). Dans une seconde section, la solution proposée est présentée. La troisième section s'intéresse à la compilation des regex. Dans la quatrième section, le flot d'implémentation est développé. La cinquième section présente les résultats d'implémentation du système. Finalement, la sixième et dernière section discute des avantages et inconvénients de l'approche proposée.

4.1 Les contraintes de la recherche d'expressions régulières

La recherche d'expressions régulières est un problème complexe. Celles-ci ont cependant l'avantage d'être très expressives et donc de pouvoir représenter plus de choses que des chaînes de caractères. Dans le cas des IDS, ce type d'expression permet de détecter des attaques et de gérer le polymorphisme de certaines attaques.

Cependant, afin de protéger les systèmes IDS d'attaques par déni de service, la recherche doit se faire dans un temps déterministe non dépendant de la chaîne de caractère traitée. Une attaque de déni de service consiste à surcharger un système en temps de calcul afin qu'il ne soit plus capable d'effectuer le service qu'il devrait. Les IDS sont, comme tous les systèmes sur internet, sujets à ce type d'attaque. Pour se prémunir de ce genre d'attaque, une première étape consiste à s'assurer que le temps de traitement du système soit indépendant des données à traiter.

Pour avoir un système déterministe en temps, une solution consiste à s'assurer de la capacité de traiter les données en $O(1)$ par caractère. Pour rechercher une expression régulière, deux méthodes principales existent : l'utilisation de NFA et l'utilisation de AFD. Les NFA présentent le désavantage de pouvoir requérir plus d'une opération par caractère alors que les AFD n'en requiert qu'une. Cependant, les AFD peuvent requérir beaucoup d'états et donc avoir une forte empreinte mémoire alors que les NFA ont un nombre d'états proportionnel à la taille de l'expression régulière. En outre, pour une expression de taille n, le temps de construction du NFA est $O(n)$ alors que celui du AFD est $O(2^n)$.

Dans ce chapitre, un compilateur d'expressions régulières vers VHDL est présenté. L'objectif est d'utiliser la forte parallélisation disponible sur les FPGA afin de rendre le parcours du NFA en $O(1)$.

4.2 Des modules correspondant à des structure d'un langage régulier

La section 2.3.2 présente la solution proposée par Sidhu et Prasanna pour effectuer la recherche d'expressions régulières en utilisant un FPGA [34]. Cette section présente les différents modules qui serviront à la réalisation du compilateur d'expressions régulières vers VHDL. Les modules proposés permettent le support de certains éléments du langage régulier présenté dans la section 2.1.2.

4.2.1 Les différentes implémentations pour la validation des caractères

Un compilateur permet, comme expliqué dans la section 2.1.3, de traduire un programme d'un langage vers un autre [50]. Dans le cas du compilateur réalisé ici, l'objectif est de convertir une expression régulière vers le langage VHDL en implémentant le NFA résultant de l'expression. La figure 4.1 présente le bloc générique servant de base pour la construction de chacun des états d'un NFA. Cette architecture est par la suite dérivée afin de supporter différents opérateurs des regex.

L'architecture présentée dans la figure 4.1 représente une implémentation générale d'un NFA. Cette architecture est composée de 5 éléments principaux : les bus `InputChar` et `État`, les blocs `Logique de caractères` et `Logique d'états` et les bascules `D`. La sortie des bascules `D` représente un état du NFA. L'activation de chaque bascule s'effectue lorsque les deux blocs `Logique de caractères` et `Logique d'états` sont actifs. Le bloc `Logique de caractères` prend en entrée le bus `InputChar` et vérifie si le caractère permet d'activer l'état actuel. Le bloc `Logique d'états` prend en entrée certains états du bus `État` et valide si ceux-ci permettent d'activer l'état auquel le bloc est relié. Le bus `État` est construit en prenant la sortie de toutes les bascules. Le bus `InputChar` est l'entrée qui transmet les caractères à chacun des blocs `Logique de caractères`. Finalement le système génère en sortie un signal `Match` indiquant si au moins un état final est validé. Ce signal est généré dans le bloc `ML` qui effectue un `OU` logique sur tous les états finaux.

Un caractère : La comparaison d'un caractère consiste à remplacer le bloc de logique `Logique de caractères` par un comparateur comme présenté dans la figure 4.2 pour le cas de l'expression *ab*. Dans cette figure, un état est généré pour chacun des caractères.

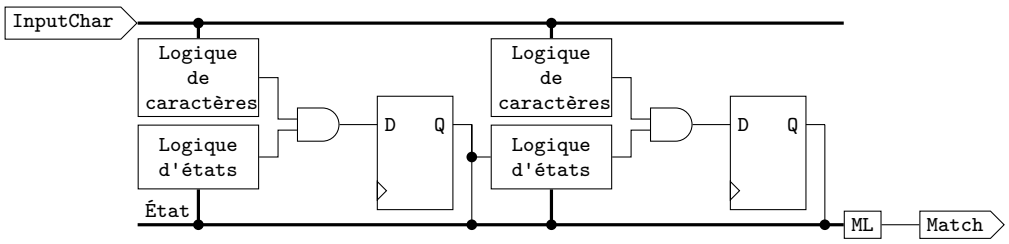


Figure 4.1 Architecture générale pour l'implémentation d'un NFA

L'activation du premier état ne dépend que du résultat du comparateur. Ceci vient du fait que dans ce cas, à chaque consommation d'un caractère, cet état peut être activé. Par exemple, si l'on reçoit la chaîne *aaaaaaaaaab*, l'état 1 serait activé pour chaque `a` reçu.

Liste de caractères : Les plages de caractère sont représentées par l'expression `[liste]`. L'élément `liste` peut contenir n'importe quelle liste de caractères. Il peut s'agir d'une plage par exemple `[a-z]` pour les caractères allant de `a` à `z`. Il est également possible d'indiquer certains caractères. Le cas d'une liste de caractères indépendants est traité par une succession de comparateurs tel que montré dans la figure 4.3a. Dans le cas d'une plage, deux comparateurs d'inégalité sont utilisés comme montré dans la figure 4.3b. Une combinaison des deux solutions peut être utilisée pour des listes plus variées telles que `[a-zBL]`.

4.2.2 Les caractères spéciaux `*` `+` `?` et `|`

Cette section présente les métacaractères Kleene (`*`), plus (`+`), interrogation (`?`) et OU (`|`). Ces caractères sont les opérateurs de base des expressions régulières ; toute expression régulière peut donc être représentée avec ces caractères. La figure 4.4 montre le NFA résultant de l'expression `na*(b|c)+d?m` et est utilisée comme exemple pour expliquer les structures matérielles des opérateurs présentés ci-dessous. Comme présenté dans la section 2.1.4, les transitions ε ne consomment pas de caractères. Dans les quatre premières sous sections, les opérateurs sont présentés. La dernière sous-section présente l'architecture matérielle pour l'implémentation du NFA de la figure 4.4.

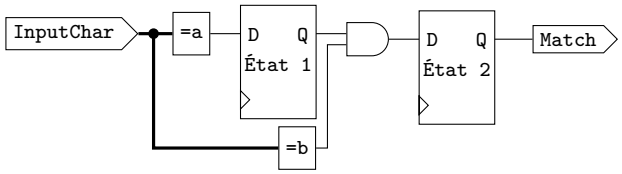
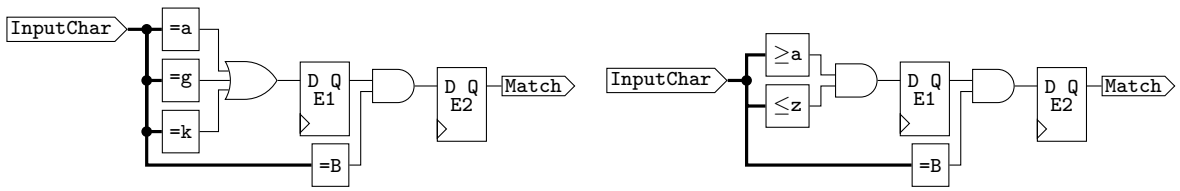


Figure 4.2 Architecture pour `ab`



(a) `[agk]` (b) `[a-z]B`

Figure 4.3 Architectures matérielle pour la comparaison de liste de caractères

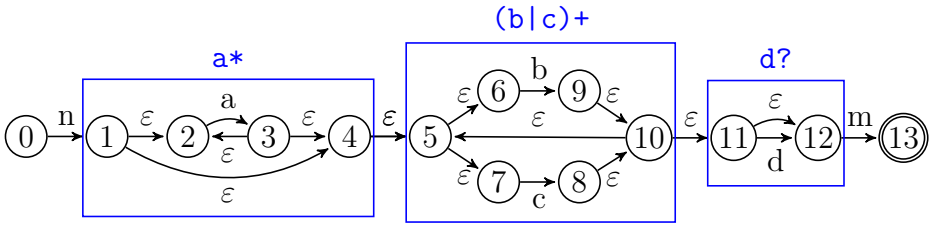


Figure 4.4 Automate résultant de l'expression `na*(b|c)+d?m`

L'opérateur Kleene (*) : L'opérateur Kleene indique que l'expression le précédant peut être présente 0 ou plus de fois. Par exemple, les chaînes de caractères *nm, nam, naam …* valident l'expression `na*m`.

Dans la figure 4.4 cet opérateur est appliqué à l'expression `a`. Les états concernés par cet opérateur sont les états 1 à 4. La transition vide (ε) de l'état 1 vers le 4 est la transition qui permet de supporter l'absence de l'expression. La transition vide de l'état 3 vers le 2 donne la possibilité de répéter l'expression. La transition de l'état 1 vers le 2 teste la présence du caractère. Ces transitions impliquent que lorsque les états 1 ou 3 sont actifs, les états 2 et 4 doivent être considérés.

Dans la figure 4.5, l'opérateur est implémenté par deux `OU` logiques. Le premier prend les états 1 et 2 en entrée et sa sortie permet de déterminer l'état 2. Ce premier `OU` permet d'implémenter les transitions ε des états 3 et 1 vers l'état 2 de la figure 4.4. Le second est mis en sortie de l'état 2 et prend en entrée les états 1 et 2. ces entrées permettent de représenter les transitions ε de la figure 4.4 des états 1 et 3 vers l'état 4.

L'opérateur Plus (+) : L'opérateur "Plus" indique que l'expression doit être présente au moins une fois. Par exemple, les chaînes *nam, naam, naaam* valident l'expression `na+m`. Cependant la chaîne *nm* ne valide pas cette expression.

Dans le cas de l'automate présenté à la figure 4.4, cet opérateur s'applique sur l'expression `(b|c)`. Les états concernés dans la figure par cet opérateur sont les états 5, 8, 9 et 10. Les

états 6 et 7 sont activés si l'état 5 est actif, l'état 10 ne s'active qu'après activation des états 8 ou 9. Si l'état 10 est actif alors l'état 5 s'active de nouveau. Les états 5 à 8 se réfèrent à l'opérateur | détaillé ci-dessous.

Dans la figure 4.5, l'opérateur `+` est implémenté par le `OU` logique précédent l'entrée des états 3 et 4. Ce `OU` logique prends les états 1, 2, 3 et 4 en entrée et sa sortie permet de déterminer les états 3 et 4. Les entrées pour les états 1 et 2 sont pour l'opérateur Kleene décrit précédemment, les entrées pour les états 4 et 5 permettent d'appliquer le `+` sur le résultat du `|` présenté ci-dessous. Ce `OU` permet d'implémenter la transition ε de l'état 10 vers le 5 de la figure 4.4.

L'opérateur OU (`|`) : La présence de deux expressions peut être représentée avec l'opérateur OU. Par exemple, les chaînes *nam* et *nbm* valident l'expression `n(a|b)m`.

Dans l'exemple donné dans la figure 4.4, cet opérateur teste l'expression `b` ou `c`. Dans l'exemple, ce sont les états 4 à 9 qui permettent de contrôler cette recherche. Les transitions vides de l'état 5 vers les 6 et 7 génèrent les deux possibilités. Cela implique que lors de l'activation de l'état 5, il faut considérer les états 6 et 7 et que l'état 10 est actif lorsque les états 6 ou 7 sont activés.

Dans la figure 4.5, l'opérateur `|` est implémenté par le `OU` logique suivant les états 3 et 4. Ce `OU` logique prends les états 3 et 4 en entrée et sa sortie permet de déterminer l'état 5. Ce `OU` permet d'implémenter les transitions ε des états 8 et 9 vers le 10 du NFA de la figure 4.4. Le fait que les états 3 et 4 de l'implémentation prennent la même condition d'état en entrée permet de réaliser les transitions ε de l'état 5 vers les 6 et 7 du NFA de la figure 4.4.

L'opérateur interrogation (`?`) : L'opérateur interrogation `?` indique que l'expression peut apparaître une fois ou ne pas apparaître. Par exemple, l'expression `na?m` valide les chaînes *nm* et *nam*.

Dans la figure 4.4, cet opérateur s'applique pour `d` et est représenté par les états 11 et 12. La présence ou non est gérée par les deux transitions ε et d, entre les états 11 et 12.

Dans la figure 4.5, l'opérateur `?` est implémenté par le `OU` logique entre les états 5 et 6. Ce `OU` permet d'implémenter la transition ε de l'état 11 vers le 12 du NFA de la figure 4.4. L'état 6 représente un état final et sa sortie est connectée au port de sortie `match`.

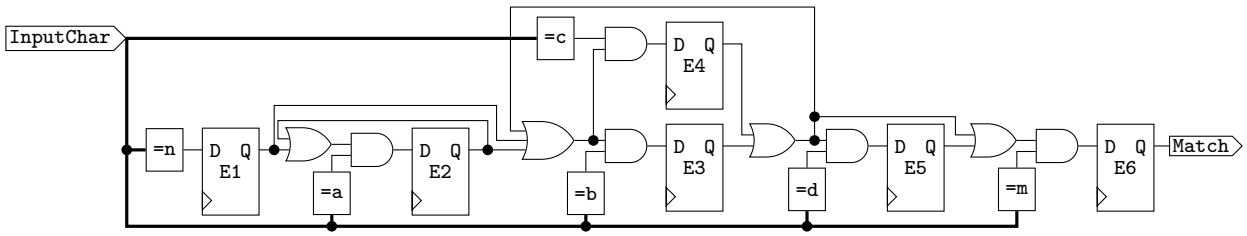


Figure 4.5 Architecture pour l'expression `na*(b|c)+d?m`

4.2.3 Compteurs

Les compteurs permettent d'agréger des expressions afin de simplifier une expression. Par exemple la chaîne `aaaa+` peut s'exprimer `a{4,}`. L'activation du compteur et sa remise à zéro doivent être réalisées avec beaucoup d'attention. En effet, un compteur pouvant s'appliquer à une expression entière, sa remise à zéro n'est possible que quand tous les états de l'expression sont désactivés. Son incrémentation s'effectue lorsqu'au moins un des derniers états de l'expression va-t-être activé. Le langage supporté contient trois types de compteurs décrits ci-dessous.

Compteur exact : Le compteur exact correspond à l'expression $\{n\}$ avec $n \in \mathbb{N}$. Ce type de compteur permet de valider si une expression a été trouvée n fois. Un exemple pour ce type de compteur est donné dans la figure 4.6 pour l'expression $ab\{n\}c$. Dans cet exemple, le compteur est appliqué à b, les conditions d'incrément et de remise à zéro dépendent donc du futur statut de l'*état 2*.

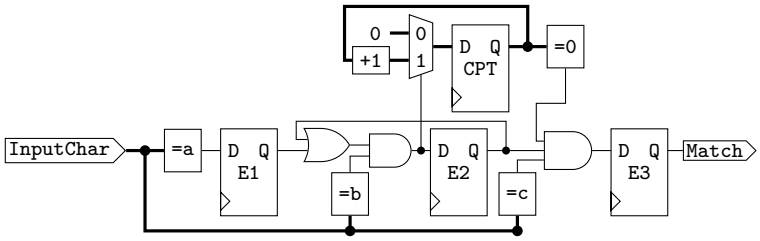


Figure 4.6 Architecture pour l'expression $ab\{n\}c$

Compteur minimum : Le compteur minimum indique si l'expression est apparue un certain nombre de fois. La figure 4.7 présente ce compteur dans le cas de l'expression $ab\{n,\}c$. Ce compteur maintien la valeur tant que les états de l'expression sont actifs et qu'il a dépassé la valeur n attendue.

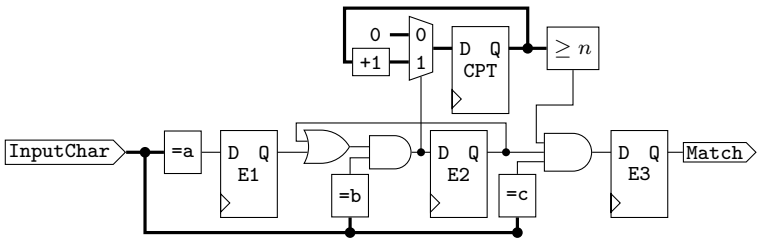


Figure 4.7 Architecture pour l'expression *ab{n,}c*

Compteur plage : Le compteur plage indique si l'expression est de n à m fois avec $(n, m) \in \mathbb{N}^2$. La figure 4.7 montre une implémentation de ce type de compteur pour l'expression *ab{n,m}c*. Ce type de compteur se réinitialise une fois la valeur m atteinte. En effet, si l'on a plus de m fois la valeur alors l'expression n'est plus valide.

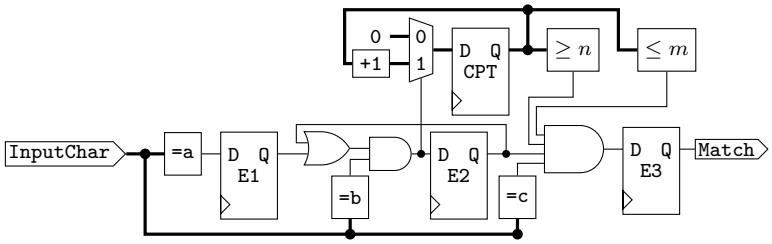


Figure 4.8 Architecture pour l'expression *ab{n,m}c*

4.3 Compilation des expressions régulières

Dans la section précédente, les différentes implémentations matérielles des éléments du langage régulier ont été présentées. Dans cette section, le compilateur est développé. Le lecteur est invité à se référer à la section 2.1.3 pour de plus amples informations sur le fonctionnement d'un compilateur. Les éléments supportés par le compilateur proposé sont les caractères, les listes de caractère, les compteurs, et les opérateurs de répétition. Dans un premier temps, une représentation forme étendue de Backus-Naur — *Extended Backus-Naur Form* (EBNF) d'une expression régulière est présentée. Par la suite, le processus général de compilation est expliqué puis différentes fonctions sont décrites.

4.3.1 Représentation EBNF

La notation en EBNF d'une regex est donnée dans la figure 4.9. Une regex `<regex>` est composée d'une ou plusieurs `<branche>` séparées par des `|`. Une `<branche>` se compose d'au moins une particule. Une `<particule>` est composée d'un `<atome>` qui peut être suivi

d'un <opérateur>. Un <atome> est composé soit d'une <regex> entre parenthèses ou d'un <jeton>, qui peut être un caractère ou une liste de caractères telle que [a-z]. Finalement, un opérateur est un des opérateurs suivants *, +, ? ou un compteur.

Par exemple, l'expression ab|c+|[efg] se décompose en trois branches : ab, c+, [efg]. La première branche possède deux particules étant les atomes a et b, la deuxième branche possède une particule c+ contenant un atome c et l'opérateur +. La dernière branche est composée d'une seule particule composée de l'atome [efg].

4.3.2 Processus de compilation d'une expression régulière

La figure 4.10 présente le processus global pour la compilation d'une expression régulière. Dans cette figure, les traits pleins indiquent un appel à la fonction et les traits pointillés indiquent un retour. Le processus s'initialise en passant en entrée une expression régulière à compiler. La première fonction appelée est la fonction ParseRegex qui par la suite appelle la fonction ParseBranche, qui appelle ParseParticule pour finalement appeler ParseAtome. Le caractère courant, car, est modifié chaque fois qu'il est nécessaire.

4.3.3 ParseRegex

La fonction ParseRegex traite les regex et son pseudo-code est donné dans la figure 4.11. Cette fonction prend en argument une liste d'états prevState et retourne deux listes d'états, debut et fin. La fonction appelle une première fois ParseBranche en lui passant en paramètre prevState jusqu'à ce que caractèreCourant soit différent de |. Elle ajoute ensuite à debut les états de tmpa et à fin les états de tmpb retournés par ParseBranche. Finalement, elle retourne debut et fin.

4.3.4 ParseBranche

La fonction ParseBranche traite les branches. La figure 4.12 présente son pseudo-code. Cette fonction prend en argument une liste d'états prevState et retourne deux listes d'états, debut, fin. Dans un premier temps, ParseParticule est appelée en passant en paramètre prevState. Les valeurs de retour sont mises dans debut pour le retour de debut et prev pour le retour de fin. La fonction est ensuite exécutée tant que l'on est dans une branche, à savoir que caractèreCourant n'est pas égal à |,) et qu'il y a encore des caractères. À chaque tour de boucle, la fonction ParseParticule est appelée et prend en argument prev. Seule la valeur de retour fin est récupérée et est enregistrée dans prev. On assigne ensuite prev à fin La fonction retourne Début et Fin.

⟨*regex*⟩ ::= ⟨*branche*⟩ { '|' ⟨*branche*⟩ }

⟨*branche*⟩ ::= ⟨*particule*⟩ { ⟨*particule*⟩ }

⟨*particule*⟩ ::= ⟨*atome*⟩ [⟨*opérateur*⟩]

⟨*opérateur*⟩ ::= '*' | '+' | '?'
| '{' ⟨*entier*⟩ '}'
| '{' ⟨*entier*⟩ ',' '}'
| '{' ⟨*entier*⟩','⟨*entier*⟩ '}'

⟨*atome*⟩ ::= '(' ⟨*regex*⟩ ')' | ⟨*jeton*⟩

⟨*jeton*⟩ ::= ⟨*caractère*⟩ | '[' ⟨*liste de caractères*⟩ ']'

Figure 4.9 Représentation EBNF d'une regex

Sigles	Signification
PR	ParseRegex
PB	ParseBranche
PP	ParseParticule
PA	ParseAtome
car	caractère courant

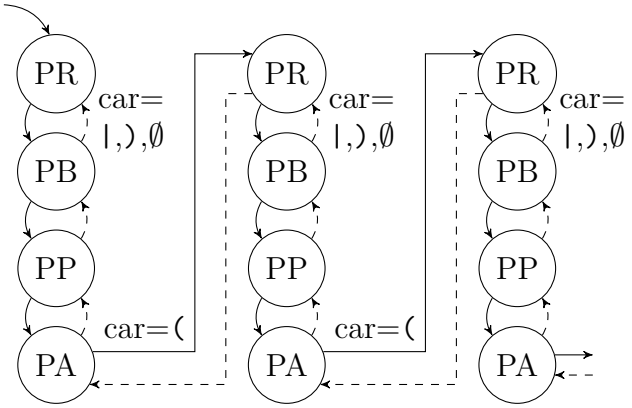


Figure 4.10 Diagramme de flux pour la compilation d'une expression régulière

4.3.5 `ParseParticule`

La fonction `ParseParticule` traite les particules. Le pseudo-code de cette fonction est présenté dans la figure 4.13. Cette fonction prend en argument une liste d'états `prevState` et retourne deux listes d'états, `Début` et `Fin`. Dans un premier temps, la `ParseAtome` est appelée en donnant en argument `prevState` et les valeurs `Début, Fin` sont récupérées. Par la suite, si `catactèreCourant='+'` alors tous les états de `debut` se voient ajouter tous les états de `fin` comme état précédent possible. Dans le cas ou `catactèreCourant='*'` alors on fait les mêmes opérations que pour le plus et on ajoute les états de `prevState` à `Fin`. Si `catactèreCourant='?'` alors on ajoute `prevState` à `fin`. Finalement, lorsque `caractèreCourant='{'` alors on crée le compteur avec la fonction `GénérerCompteur`. On

```
   Entrées : prevState
   Sorties : Début, Fin
1  Fonction ParseRegex(prevState) /* fonction pour le parsage d'une regex
    */
2  |  Début =nouvelle liste ;
3  |  Fin =nouvelle liste ;
4  |  répéter
5  |  |  tmpa, tmpb=ParseBranche(prevState) ;
6  |  |  ajouter tmpa à Début ;
7  |  |  ajouter tmpb à Fin ;
8  |  jusqu'à caractèreCourant ≠ '|' ;
9  |  retourner Début, Fin ;
```

Figure 4.11 Fonction ParseRegex

```
   Entrées : prevState
   Sorties : Début, Fin
1  Fonction ParseBranche(prevState) /* fonction pour le parsage d'une
    branche                                                          */
2  |  Début, tmpPrev =ParseParticule(prevState) ;
3  |  tant que caractèreCourant ∉ {'|', ')', ∅} faire
4  |  |  ___, Fin =ParseParticule(prevState) ;
   |  |  /* On ignore la première valeur retournée par ParseParticule   */
5  |  retourner Début, Fin ;
```

Figure 4.12 Fonction ParseBranche

assigne à tous les états de `Fin` le compteur créé. Enfin, on ajoute à tous les états de `Début` tous les états de `Fin` comme états précédents.

4.3.6 `ParseAtome`

La fonction `ParseAtome` traite les atomes, qui peuvent être soit un jeton ou une expression régulière. Son pseudo-code est présenté dans la figure 4.14. Cette fonction prend en entrée une liste d'états `prevState` et retourne deux listes d'états, `debut` et `fin`. Si la condition `caractèreCourant='('` est validée alors `ParseRegex` est appelé avec le paramètre `prevState`. Les listes d'états retournées par `ParseRegex` sont enregistrées dans `Début` et `Fin` . Dans les autres cas, la fonction génère l'état en déterminant le jeton actuel, elle ajoute à `ÉtatTemp` les états précédents qui sont contenus dans `prevState`. Elle assigne ensuite `ÉtatTemp` à `Début` et `Fin`. Finalement, la fonction retourne `Début` et `Fin`

```
   Entrées : prevState
   Sorties : Début, Fin
 1 Fonction ParseParticule(prevState)          // traitement d'une particule
 2 |  Début, tmpPrev =ParseAtome(prevState) ;
 3 |  suivant caractèreCourant faire
 4 |  |  cas où ' * ' faire
 5 |  |  |  Ajouter à tous les états de Début les états de Fin comme état précédent ;
 6 |  |  |  Ajouter à la liste Fin tous les état de prevState ;
 7 |  |  cas où ' + ' faire
 8 |  |  |  Ajouter à tous les états de Début les états de Fin comme état précédent ;
 9 |  |  cas où '?' faire
10 |  |  |  Ajouter à la liste Fin tous les état de prevState ;
11 |  |  cas où '{' faire                      // cas ou l'on a un compteur
12 |  |  |  Compteur =GénérerCompteur() ;
13 |  |  |  Ajouter Compteur tous les états de Fin ;
14 |  |  |  Ajouter à tous les états de Début les états de Fin comme état précédent ;
15 |  retourner Début, Fin ;
```

Figure 4.13 Fonction ParseParticule

4.3.7 Les différentes variables et types utilisés

Dans cette section, certaines variables et types particuliers utilisés pour la réalisation de l'algorithme des figures 4.11, 4.12, 4.13 et 4.14 sont présentés. Le NFA est représenté par une liste contenant tous les états. Celle-ci n'est pas explicitée dans les figures, mais chaque état créé est ajouté à cette liste. Les états de cette liste sont de type `State` qui est une structure qui contient tous les états précédents d'un état ainsi que le jeton qui s'applique à l'état. Le jeton représente soit une `liste` de caractères ou un unique `caractère`. Il y a deux variables globales dans l'algorithme qui sont `CaractèreCourant` et `Regex`. `Regex` est la variable contenant l'expression. `CaractèreCourant` donne le caractère actuellement traité, celui-ci est extrait de `Regex` à chaque fois que nécessaire.

4.4 Flot de synthèse

L'objectif premier du compilateur est de permettre une automatisation du processus pour permettre l'implémentation d'un jeu d'expressions régulières sur un FPGA. Ainsi, le flot présenté dans la figure 4.15 présente le flot complet du système.

```
Entrées : prevState
Sorties : Début, Fin
1 Fonction ParseParticule(prevState) // fonction pour le parsage d'un
   atome
2     si caractèreCourant = '(' alors
3         Début,Fin = ParseRegex(prevState) ;
4     sinon
5         Jeton = GénérerJeton() ;
6         ÉtatTemp =GénérerÉtat(prevState, Jeton) ;
7         Début =ÉtatTemp ;
8         Fin =ÉtatTemp ;
9     retourner Début, Fin ;
```

Figure 4.14 Fonction ParseAtome

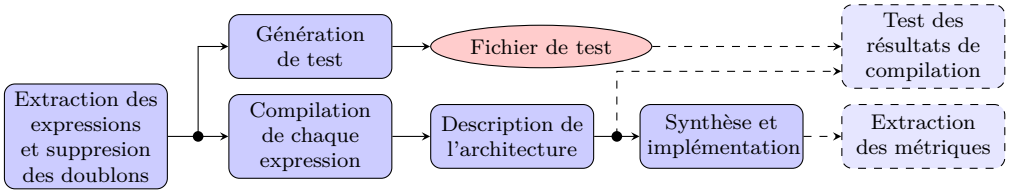


Figure 4.15 Flot de synthèse d'un jeu de regex sur un FPGA

4.4.1 Présentation du flot

Dans la première étape, les regex à implémenter sont extraites. Lors de cette extraction, le système supprime les expressions en doubles. Par la suite, les expressions sont compilées vers des fichiers VHDL indépendants, et cette étape est présentée dans la section 4.3. Ce choix permet de rendre plus facilement réutilisables les résultats de compilation. Le système permet de générer des tests afin de valider les différents modules compilés. Ces tests sont toutefois réalisés séparément. Finalement, la synthèse et l'implémentation sont effectuées une fois tous les fichiers compilés et intégrés dans l'architecture principale.

4.4.2 Génération des tests

Précédemment, le flot complet du système a été présenté. Dans cette section, les mécanismes de génération des tests sont présentés. Pour tester si une architecture compilée est valide, il faut vérifier que le système détecte les expressions régulières correctement. La génération est faite de telle sorte que tous les états possibles de l'automate soient activés au moins une

fois. Le système de génération valide également les opérateurs de répétition en prenant les cas limites.

Pour les opérateurs plus (`+`) et Kleene (`*`), les cas générés contiennent l'apparition de l'expression 0, 1 et n fois au minimum. La valeur de n est arbitrairement définie. Par exemple, pour l'expression `ab+`, les chaînes générées sont `a`, `ab`, `abbbb` si $n = 4$.

Les compteurs exact et minimum sont testés autour du domaine de validité. Ainsi la valeur exacte est vérifiée ainsi que les cas où on la dépasse et où on ne l'atteint pas. Par exemple, pour l'expression `ab4`, les chaînes générées sont `abbb`, `abbbb` et `abbbbb`.

Les compteurs plage sont testés autour des domaines de validité des valeurs minimales et maximales ainsi qu'au moins une valeur aléatoire comprise dans l'intervalle. Par exemple, pour l'expression `ab{4,10}`, les chaînes générées pourraient être `abbb` (3 b), `abbbb` (4 b), `abbbbb` (5 b), `abbbbbbbbb` (9 b), `abbbbbbbbbb` (10 b), `abbbbbbbbbbb` (11 b) ainsi que `abbbbbbb` (7 b).

Finalement, les listes de caractères sont le dernier élément à ne pouvoir être vérifié de manière exhaustive. En effet, pour une expression telle que `[ab]5`, il y a 2^5 cas possibles. La vérification se fait donc en générant des valeurs aléatoires comprises dans l'intervalle des caractères possibles.

Toutes les chaînes de test sont par la suite mises en commun pour générer un flux complet. Le système ajoute également des séquences de caractère aléatoire entre chaque chaîne pour rendre le test plus robuste. Enfin les positions des différentes correspondances possibles sont déterminées en utilisant la librairie Python de recherche des expressions régulières.

4.4.3 Réalisation des tests

Les tests ne sont pas effectués automatiquement, mais les éléments nécessaires à leur exécution sont générés pour chaque expression compilée. Leur exécution s'effectue sur un banc d'essai codé en VHDL. Ce banc d'essai implémente l'architecture compilée pour l'expression et lit le flux de caractère contenant les cas de test. Enfin, les positions de correspondante sont comparées avec les positions générées par la librairie Python.

4.4.4 Synthèse et implémentation

Une fois les expressions compilées, il faut réaliser la synthèse et l'implémentation du système. Cette étape du processus requiert une intervention humaine. Puisque, le système ne se destine pas à un FPGA particulier, il est de la responsabilité de l'utilisateur de définir les différentes

conditions d'implémentations telles que les ports d'entrées sorties ou la période d'horloge désirée. L'intégration des différentes expressions se fait en implémentant une architecture complète contenant tous les circuits compilés. Ce système possède trois entrés et une sortie. Les trois entrées sont le signal d'horloge, le signal de remise à zéro et le bus d'entrée de caractère. La sortie est un bus intégrant toutes les sorties de correspondance de tous les éléments.

4.4.5 Extraction des métriques

Une fois la synthèse et l'implémentation réalisées, les métriques extraites sont le nombre de LUT, le nombre de registres et la période minimale d'horloge. Cette extraction peut se faire à différents moments du processus en fonction des possibilités offertes par le logiciel de synthèse et d'implémentation.

4.5 Implémentation et résultats

La section précédente présentait les différents modules dont est composée l'architecture. Dans cette section, les résultats d'implémentation sur un FPGA *xc7z020* sont présentés. Ces résultats sont donnés dans le tableau 4.1. Ils contiennent le résultat d'implémentation de deux expressions uniques ainsi que de trois jeux d'expressions, deux de 250 expressions et un de 524. Toutes les expressions régulières ont été extraites de Snort. Quatre éléments sont mesurés : le nombre d'états estimé grâce au compilateur, le nombre de LUT et de registres utilisés et la période d'horloge obtenue.

La première expression possède 21 états après compilation et utilise 21 registres, la seconde a 14 états et requiert 13 registres. Les LUT requises pour ces deux expressions sont au nombre de 26 pour la première et de 15 pour la seconde. Les deux expressions ont une période d'horloge équivalente à 2 ns. Comme l'architecture utilise un encodage à une bascule par état (*one-hot*), le nombre de registres devrait être égal au nombre d'états. La variation constatée pour la seconde expression s'explique par l'optimisation du synthétiseur pour les branches `REQIMG` et `RVWCFG` qui ont leur premier `R` en commun.

L'implémentation des jeux d'expressions régulières montre également cette optimisation. Pour les jeux 1 et 2 de 250 expressions, on a respectivement 3071 et 3073 états ainsi que 2541 et 2595 registres. Dans le cas du jeu 3, ayant 524 expressions, on constate 5541 registres pour 6227 états. Il est intéressant de noter que le nombre de LUT n'a pas de tendance particulière avec le nombre d'états. En effet, pour le cas du jeu 1, 3255 LUT sont requises alors 2728 sont utilisées pour le jeu 2.

Enfin, la période d'horloge pour le jeu 1 est de 2,5 ns, inférieur de moitié de celle du jeu 2 bien que le nombre d'éléments logiques nécessaires pour le jeu 2 soit inférieur au jeu 1. Ceci s'explique par une longueur de fils plus importante pour certaines expressions du jeu 2.

Tableau 4.1 Résultats d'implémentation de la compilation d'expression régulières

Expression	États	LUT	Registres	Période(ns)
`template=(https?|ftps?|php)`	21	26	21	2
`\x3c(REQIMG|RVWCFG)\x3e)`	14	15	13	2
Jeu 1 de 250 expressions	3071	3255	2541	2.5
Jeu 2 de 250 expressions	3073	2728	2595	5
Jeu 3c de 524 expressions	6227	6782	5541	5

4.6 Discussion

Ce chapitre présente un compilateur de regex permettant de simplifier le flot de synthèse et de permettre une plus simple utilisation des FPGA pour la recherche de celles-ci. Ce système permet de parcourir un NFA dans un temps $O(n)$ où n est le nombre d'états qui est proportionnel à la taille des regex. Cette section présentera tout d'abord les avantages de l'utilisation du compilateur. Par la suite, certains inconvénients sont exposés. Finalement, l'intérêt de ce compilateur pour d'autres applications est discuté.

La solution proposée dans ce chapitre possède trois avantages principaux. Le premier est une simplification et une automatisation du processus de génération de code VHDL et donc du temps de mise en production. Le second intérêt est la possibilité de parcourir un NFA en $O(1)$ par caractère. Finalement, le troisième avantage est la diminution de la mémoire nécessaire pour la recherche d'une expression par rapport à l'utilisation d'un AFD.

Malgré les avantages certains du compilateur proposé, certains inconvénients sont présents. Le premier est le non-support de certains des éléments des expressions régulières compatibles avec Perl — *Perl Compatible Regular Expression* (PCRE). Seuls les langages réguliers au sens de Chomsky sont supportés contrairement au travail de Mitra et al. [35]. Toutefois, contrairement au travail proposé par Sidhu et Prasanna [34], les compteurs sont supportés ce qui permet de réduire la quantité de ressources nécessaires pour les expressions les utilisant. Le second inconvénient est le temps pour la réalisation des mises à jour. Le système proposé améliore l'utilisation des ressources, mais les mises à jour partielles sont plus complexes. De plus, la reconfiguration d'un FPGA peut prendre plusieurs millisecondes [51] ce qui, dans les applications réseau, peut être un problème majeur. Enfin, la solution proposée implique la recherche de toutes les expressions régulières possibles à chaque fois. Cependant, dans le

cas des IDS, il existe un filtre pour déterminer quelle expression doit être cherchée. Ainsi, la recherche de toutes les expressions est une perte des ressources.

Comme pour Sidhu et Prasanna et Mitra et al. le système est capable de rechercher un ensemble d'expressions en parallèle, et ce avec un temps maximal égal au temps pour la recherche de la plus grande expression ou à la longueur du flux d'entrée. Un inconvénient majeur, pour nos travaux comme pour les précédents, est lié à la mise à jour du système. Ces solutions de recherche d'expressions régulières sont donc intéressantes pour des applications n'ayant pas besoin de faire souvent des mises à jour.

Suite à l'étude du travail présenté dans ce chapitre, on remarque que la solution générée, comme pour les travaux de Mitra et al. et de Sidhu et Prasanna, requiert la recherche de toutes les expressions. Comme noté précédemment, le temps de mise à jours est aussi une contrainte importante. Une solution a été proposée par Divyasree et al. [44]. Dans cette solution, l'architecture supporte les mises à jour car les différents blocs de l'architecture sont reconfigurables. Toutefois la solution proposé par Divyasree et al. a un temps de reconfiguration qui permet des attaques par déni de service. Ces constatations ont amené à la réalisation présentée dans le chapitre 5.

CHAPITRE 5 UNE ARCHITECTURE INTERMÉDIAIRE À GROS GRAINS POUR LA RECHERCHE D'EXPRESSIONS RÉGULIÈRES

Dans le chapitre 4, un compilateur d'expression régulières vers VHDL est présenté. Le présent chapitre introduit une architecture intermédiaire à gros grains qui effectue la même fonction qui est dynamiquement reconfigurable. Cette architecture a été publiée à la conférence ACM Great Lakes Symposium on VLSI 2017 (GLSVLSI'17) [17]. Dans un premier temps, les raisons de cette architecture sont présentées. Par la suite l'architecture réalisée est décrite. Dans une troisième section, les résultats d'implémentation sont détaillés. Finalement, l'architecture est discutée.

5.1 Introduction

Trois contraintes majeures s'appliquent aux systèmes de traitement de données dans les applications réseau : temps de recherche faible, mise à jour rapide et faible consommation d'énergie. Dans le cas des IDS, l'utilisation d'un FPGA pour faire cette opération a été démontrée intéressante dans le chapitre 4. La solution proposée ne concourt pas à des mises à jour rapides et à une faible consommation d'énergie. Les mises à jour sont lentes, car elles requièrent de reconfigurer le FPGA. Dans le même temps, la consommation d'énergie est importante, car pour valider une expression régulière, toutes les expressions du jeu sont recherchées en parallèle.

Qui plus est, le processus de Snort ne requiert pas la recherche de toutes les expressions régulières en même temps. Comme l'illustre la figure 5.1, Snort commence par effectuer une recherche exacte de chaînes de caractères. Si une correspondance est détectée, alors le système regarde si une expression régulière doit être recherchée. Si tel est le cas alors l'engin de recherche d'expression régulière est lancé, sinon une alerte est levée. Si une correspondance est trouvée avec l'expression régulière à chercher alors une alerte est levée.

Étant donné qu'une seule expression régulière doit être recherchée, un moyen de réduire la consommation d'énergie du système est de faire uniquement la recherche de cette expression sur du matériel spécialisé. Cependant, un tel système doit pouvoir être configuré le plus rapidement possible. Afin de passer rapidement d'une expression à une autre, une solution consiste à utiliser des systèmes multi-contextes qui ont l'avantage d'être à même de sélectionner une configuration dans un temps très court [51]. Pour supporter des mises à jour rapides

des règles, une solution consiste à utiliser une architecture à gros grains qui est plus rapide à configurer que des architectures à grains fins [52].

La solution retenue est une architecture intermédiaire à gros grains. Les architectures intermédiaires sont des architectures reconfigurables implémentées sur FPGA [45].

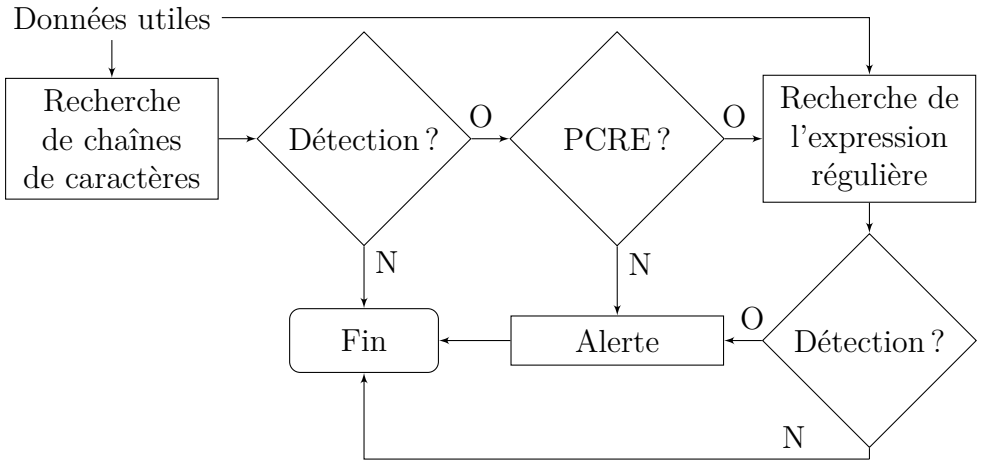


Figure 5.1 Processus de Snort

5.2 Description de l'architecture

La section précédente présente les raisons de la réalisation d'une architecture intermédiaire. Dans cette section l'architecture réalisée est décrite. Tout d'abord, l'architecture générale est présentée. Par la suite, le bloc de validation d'états est développé. Enfin, deux types de blocs de validation de caractères sont développés.

5.2.1 Vue d'ensemble de l'architecture

Une vue globale du circuit est montrée dans la figure 5.2. Le circuit est composé de N Éléments Reconfigurables de Détermination d'état (ERDE), chacun prenant en entrée un caractère `InputChar`, un numéro de contexte `Contexte` et m états. Les m états sont déterminés par le bloc *Sélecteur* qui prend en entrée `Contexte` et le bus `États` concaténant les N sorties des ERDE. La sortie `Correspondance`, qui indique si une correspondance a été trouvée, est générée par le bloc *StateVal* qui a en entrée les N états et `Contexte`. Le système supporte jusqu'à C contextes. Chaque contexte correspond à une configuration particulière du système (c'est-à-dire une expression régulière).

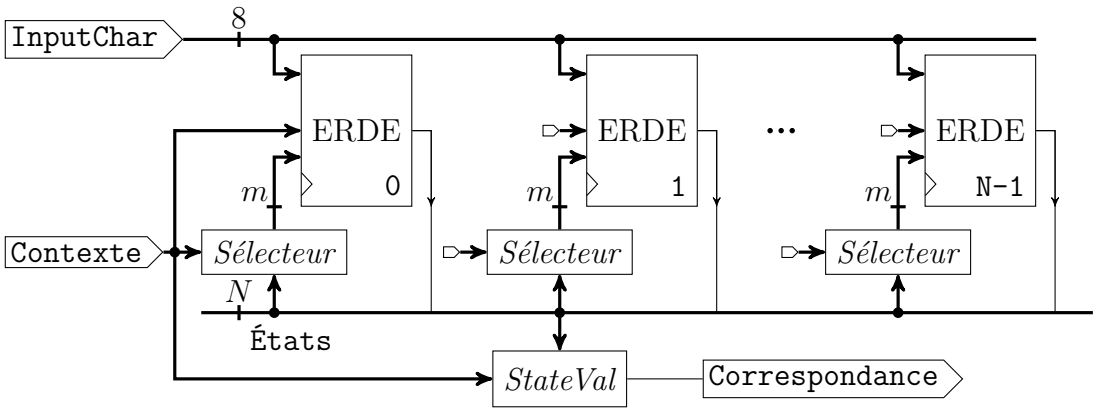


Figure 5.2 Présentation générale de l'architecture

Les ERDE sont structurés tels que présenté dans la figure 5.6. Chaque ERDE possède deux blocs : *CVal* qui effectue la validation d'un caractère et *StateVal* qui vérifie les états. Les deux blocs prennent en entrée le bus `Contexte`. Le bloc *CVal* prend également en entrée le caractère du bus `InputChar` et génère en sortie un signal indiquant si le caractère valide la condition d'activation de l'état représenté par le ERDE. Le bloc *StateVal* prend en entrée le bus `ÉtatsSel` de taille m généré par le bloc *Sélecteur*, le bloc génère un signal en sortie indiquant si les états actuellement actifs peuvent valider l'état du ERDE. Les sorties des deux blocs sont connectées aux deux entrées d'un ET logique dont la sortie est connectée à une bascule D. La sortie de la bascule D est connectée à la sortie du ERDE qui est connectée au bus `États` du système.

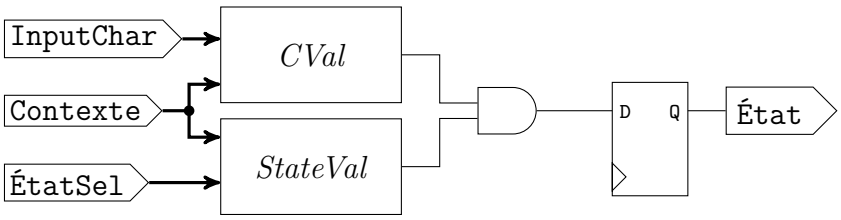


Figure 5.3 Présentation général d'une RSPE

5.2.2 Bloc de validation d'états *StateVal*

Le bloc *StateVal*, présenté dans la figure 5.4, permet de faire la validation d'états. Ce bloc est composé d'une mémoire *CMem* pouvant contenir C mots de taille m. Un ET bit à bit est appliqué à la sortie de *CMem* et à l'entée `ÉtatSel` afin de sélectionner les états intéressant dans le contexte. Un NON OU logique est appliqué à tous les bits à la sortie de *CMem* afin

de déterminer si un état précédent est nécessaire. Un OU logique est connecté à la sortie du ET et du NON OU et détermine si les états requis quand il en faut sont actifs.

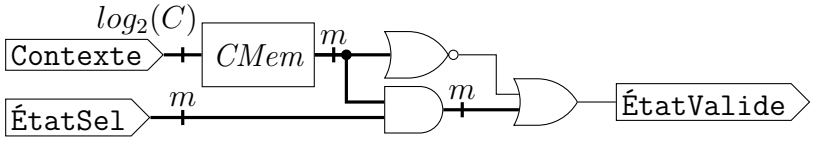


Figure 5.4 Bloc de validation d'états

5.2.3 Un design de *Sélecteur*

Le *Sélecteur* est un élément prenant un bus de taille N en entrée et qui sélectionne m fils en sortie. Un design proposé pour le *Sélecteur* est présenté à la figure 5.5. Le bloc prend en entrée *Contexte* et `États`, il génère la sortie `ÉtatSel`. Le bloc est en interne composé de m multiplexeurs. Le premier multiplexeur permet de sélectionner un fils parmi $N+1$ fils, les N du bus d'`États` et un ayant la valeur `'1'`. Le second sélecteur permet lui de sélectionner parmi N fils, $N-1$ des `États` et un avec la valeur `'1'`. Les multiplexeurs sont ajoutés, ainsi de suite jusqu'au m$^{\text{ième}}$ qui sélectionne 1 fils parmi $N-m+1$. Les fils sélectionnés par chacun des sélecteurs dépendent de la mémoire de configuration *SelMemConf* qui contient C mot dont la taille est égale à M_{SelMem} donné par l'équation 5.1.

$$M_{SelMem} = \sum_{k=N-m+1}^{N+1} \lceil \log_2(k) \rceil \tag{5.1}$$

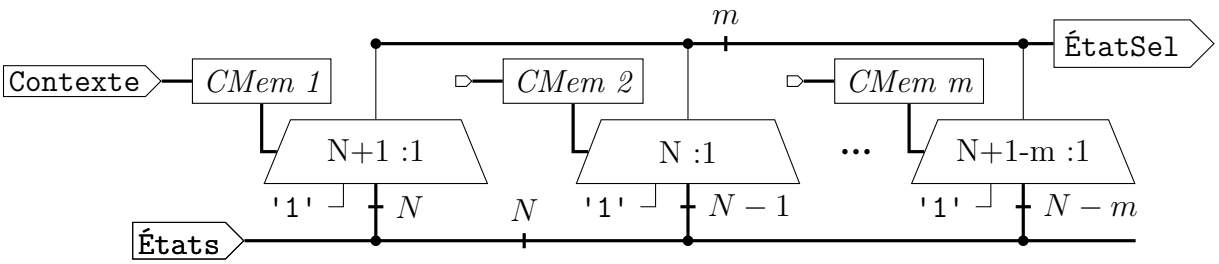


Figure 5.5 Présentation du *Sélecteur* d'états de N vers m

5.2.4 Bloc de type 1

Le bloc de type 1 permet de vérifier la présence ou l'absence d'un caractère unique. Ce bloc permet donc de valider des expressions régulières telles que `a` ou `[^a]`. Ce bloc est présenté

dans la figure 5.6. Il inclut deux mémoires pour la validation du caractère, *CharMem* et *InvMem* qui sont toutes deux connectées au bus `Contexte`. Ces deux mémoires contiennent C mots de 8 bits et 1 bit respectivement. La sortie de la mémoire *CharMem* est connectée à un comparateur qui prend également entrée `InputChar`. Un OU exclusif est ensuite appliqué aux signaux sortant du comparateur et *InvMem*. La sortie du OU exclusif indique si le caractère est valide et est connectée au ET logique du ERDE.

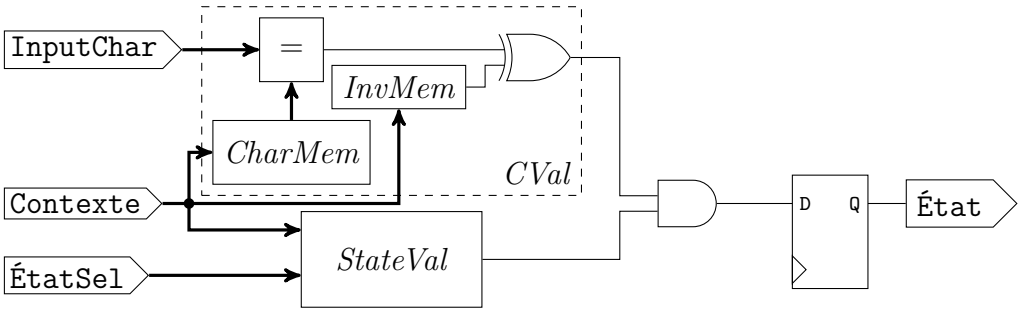


Figure 5.6 Bloc de type 1

5.2.5 Bloc de type 2

Le bloc de type 2 permet de tester jusqu'à 256 caractères, il est présenté à la figure 5.7. Ce bloc permet donc de valider des expressions telles que `[abcd]` ou `[azerty]`. Il peut également être utilisé, sous certaines conditions, pour la validation d'une expression telle que `(a|b|c|d)`. Ce bloc prend en entrée le numéro de contexte `Contexte` et un caractère. Le bus `Contexte` est connecté à la mémoire *BMCharMem* contenant C mots de 256 bits. La sortie de la mémoire est connectée à un multiplexeur 256 :1 qui permet de faire la validation du caractère. Le bus `InputChar` est connecté pour sélectionner le signal correspondant au caractère.

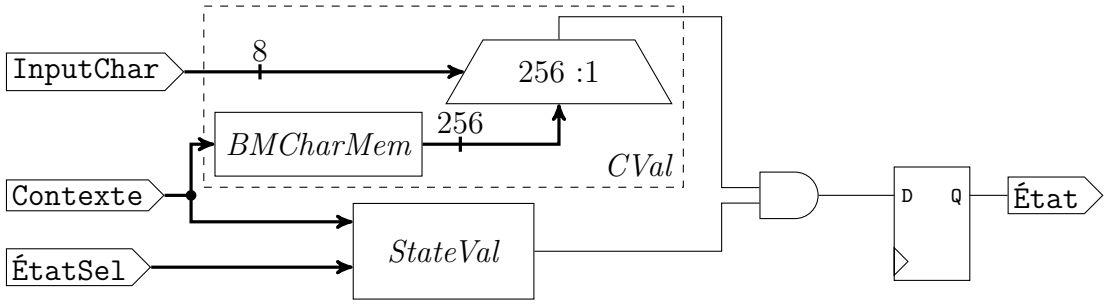


Figure 5.7 Bloc de type 2

5.3 Les aspects pratiques

La section précédente présente les différents modules de l'architecture. Cette section fait une analyse de l'architecture. Dans un premier temps, le coût des différents éléments est analysé. Par la suite, la détermination du nombre de chaque type utilisé est expliquée. Finalement les méthodes pour la modification des contextes sont explorées.

5.3.1 Coût des éléments

Comme il s'agit ici d'une architecture intermédiaire implémentée sur un FPGA, deux métriques sont considérées, l'utilisation de logique U_L et de mémoire U_M. La fonction de coût total du circuit est présentée dans l'équation 5.2. Dans cette équation, N_{T1} et N_{T2} représentent respectivement le nombre de blocs de type 1 et le nombre de blocs de type 2 avec $N = N_{T1} + N_{T2}$. Les valeurs U_{LT1} et M_{LT1} représentent l'utilisation de logique et de mémoire pour le type 1, U_{LT2} et M_{LT2} représentent ces utilisations pour le type 2. Enfin, U_{LSel} et U_{MSel} représente l'utilisation de logique et de mémoire pour le *sélecteur*.

$$\begin{bmatrix} U_L \\ U_M \end{bmatrix} = N_{T1} \times \begin{bmatrix} U_{LT1} \\ U_{MT1} \end{bmatrix} + N_{T2} \times \begin{bmatrix} U_{LT2} \\ U_{MT2} \end{bmatrix} + N \times \begin{bmatrix} U_{LSel} \\ U_{MSel} \end{bmatrix} \tag{5.2}$$

Chaque type possède le même bloc de validation d'états *StateVal*. Bien que sur un FPGA les blocs de logique et de mémoire soient prédéterminés, il est quand même possible d'estimer le coût minimum requis. Ainsi, l'utilisation de mémoire pour *StateVal* peut être estimée à $C \times m$. L'estimation du coût en logique est développée ci-dessous. La fonction logique de `ÉtatValide` est $f = \sum_{i=1}^{m} a_i b_i + \overline{\sum_{i=1}^{m} a_i}$. Avec f représentant la valeur du signal `ÉtatValide`, b_i le i$^{\text{ème}}$ bit du bus `ÉtatSel` et a_i le i$^{\text{ème}}$ bit du bus en sortie de *CMem*. Prenons la fonction $h = \sum_{i=1}^{m} a_i b_i + g$, ici h est une fonction avec $2 \times m + 1$ entrées. On défini maintenant g tel que $g = \overline{\sum_{i=1}^{m} a_i}$, il s'agit d'une fonction à m entrées. La fonction f peut donc être considérée comme une fonction avec $3 \times m + 1$ entrées. Si l'on considère maintenant le coût en logique comme étant proportionnelle au nombre d'entrées alors le coût logique de *StateVal* peut être estimé à $3 \times m + 1$.

En outre, la génération 7 des FPGA de Xilinx utilise des LUT à 6 entrées [53]. Ces LUT permettent chacune de résoudre une fonction logique de 6 entrées. Pour les fonctions plus grandes plusieurs LUT sont utilisées, ce paragraphe propose une méthode pour en estimer le nombre. Par exemple, pour réaliser un OU logique sur 12 entrées, 3 LUT sont nécessaires, comme présenté dans la figure 5.8. Deux LUT sont nécessaires chacune pour faire le OU logique sur 6 des entrées et une pour faire le OU logique sur les deux sorties des deux premières

LUT. Comme 4 entrées sont encore disponibles, l'architecture présentée permet également de résoudre un OU logique à 16 entrées, les 4 entrées supplémentaires sont connectées aux 4 entrées disponibles de la troisième LUT. Ainsi la profondeur de la cascade de LUT peut être estimée à $\lceil \log_6(Ne) \rceil$ avec Ne le nombre d'entrées de la fonction. Le nombre de LUT nécessaires pour chaque niveau de fonction est donc égal à $\lceil Ne \div 6^k \rceil$ avec k la position dans la cascade. Ainsi, au premier niveau avec 12 entrées, $12 \div 6 = 2$ LUT sont nécessaires pour traiter toutes les entrées. Au second niveau $\lceil 12 \div 6^2 \rceil = 1$ LUT est nécessaire. En appliquant le même raisonnement, on obtient l'équation 5.3 qui permet d'estimer le nombre de LUT pour des FPGA Xilinx de génération 7 pour le bloc de validation d'états.

$$N_{LUT} = \left\lceil \sum_{k=1}^{\lceil log_6(m \times 3+1) \rceil} \frac{m \times 3 + 1}{6^k} \right\rceil \tag{5.3}$$

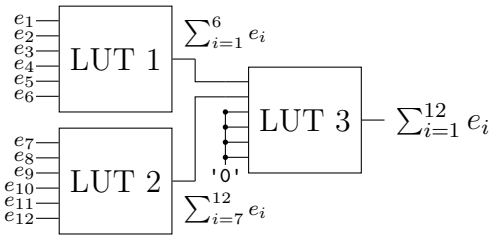


Figure 5.8 Implémentation d'un OU logique à 12 entrées avec 3 LUT à 6 entrées

Le bloc de validation de caractères du type 1 utilise 7 bits de mémoire et effectue une opération logique de 17 bits en entrée. Ainsi en ajoutant le coût du bloc *StateVal*, on peut estimer que $U_{LT1} = 8 + 8 + 1 + 2 \times m$ et $U_{MT1} = (m + 7) \times C$. Dans, le cas du type 2, *CVal* utilise 256 bits de mémoires et effectue une opération logique à $256 + 8 = 254$ entrées. On obtient donc $U_{LT2} = 264 + 2 \times m$ et $U_{MT2} = (m + 256) \times C$. Les équations 5.4 et 5.5 introduisent les fonctions $L(M)$ et $K(M)$. Ces fonctions permettent de déterminer, en fonction de m, la taille minimale d'une liste de caractères pour que les ERDE de type 2 deviennent plus intéressants que ceux du type 1. Ainsi, si l'on prend la valeur minimale entre les fonctions $L(m)$ et $K(m)$ alors celle-ci détermine le nombre minimal de caractères qu'il faut comparer pour que le type 2 soit plus économe que le type 1. Comme $\forall m \in \mathbb{N}^*, K(m) > L(m)$, le nombre minimal de caractères est déterminé en fonction de $L(m)$, tracée dans la figure 5.9. Pour $m = 1$, ce bloc est intéressant si environ 13 caractères sont à comparer. Pour $m = 20$, le bloc est avantageux lors de la comparaison de 5 caractères. L'utilisation de ce bloc pour la comparaison de listes de caractères est donc rapidement avantageuse à mesure que m augmente.

$$L(m) = \left\lceil \frac{3 \times m + 1 + 264}{3 \times m + 1 + 17} \right\rceil \tag{5.4}$$

$$K(m) = \left\lceil \frac{256 + m}{8 + m} \right\rceil \tag{5.5}$$

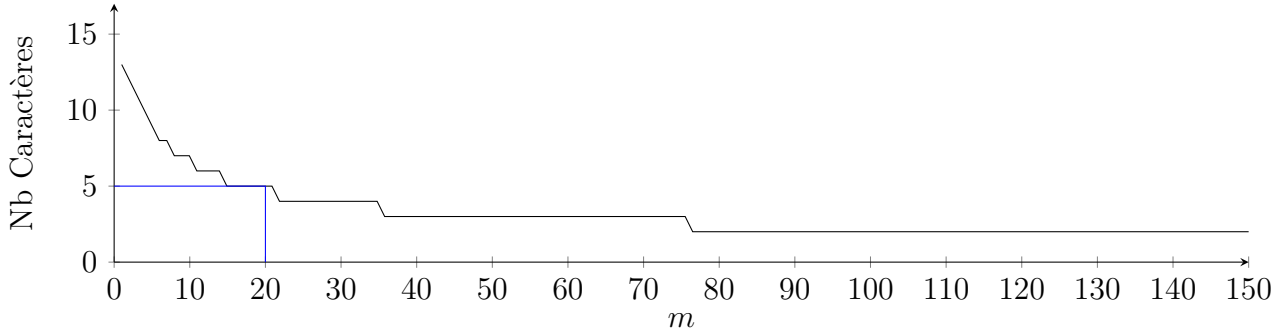


Figure 5.9 Nombre minimal de caractères à comparer en fonction de m pour l'utilisation d'un ERDE de type 2

5.3.2 Réflexion à propos de l'utilisation d'un sélecteur

Le *Sélecteur* est un élément qui doit permettre de réduire le coût en logique pour le circuit. Cependant comme le système est implémenté sur un FPGA, la logique d'implémentation peut s'avérer coûteuse. Prenons pour exemple le cas où $m = 50$ et $N = 51$, dans ce cas, le sélecteur requiert 50 multiplexeurs et une mémoire de contexte avec des mots de 249 bits. Or dans ce cas, la mémoire de validation de contexte du ERDE requiert une mémoire de 50 bits et 51 bits sans l'utilisation du sélecteur. Cette section cherche donc à estimer les cas où l'utilisation d'un sélecteur s'avère être un choix intéressant.

L'équation 5.6 présente le vecteur de coût du sélecteur. Comme pour le bloc de validation d'une liste de caractères, le coût en logique d'un multiplexeur dépend de manière linéaire de son nombre total d'entrées. Ainsi, pour un multiplexeur k :1, le nombre total d'entrées est $k + \log_2(k)$. Le coût de la mémoire est égal à $C \times M_{SelMem}$ avec M_{SelMem} défini dans l'équation 5.1.

$$\begin{bmatrix} U_{LSel} \\ U_{MSel} \end{bmatrix} = \begin{bmatrix} \sum_{k=N-m+1}^{N+1} k + \lceil \log_2(k) \rceil \\ C \times \sum_{k=N-m+1}^{N+1} \lceil \log_2(k) \rceil \end{bmatrix} \tag{5.6}$$

Pour estimer le point d'intérêt du sélecteur, il faut chercher la valeur à partir de laquelle la somme des utilisations de ressources avec le sélecteur est inférieure à celle sans ce dernier.

Cela revient à résoudre l'équation 5.7. La figure 5.10 montre les valeurs de m maximales en fonction de N pour lesquelles la différence d'utilisation de logique ou de mémoire permet de réduire le coût global.

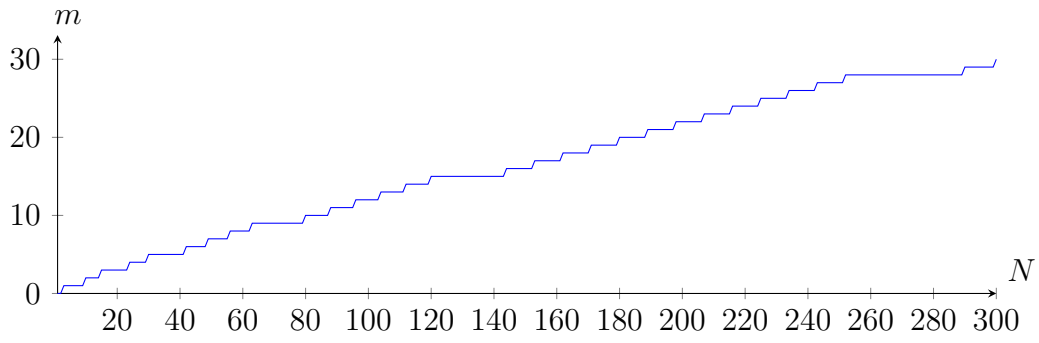
Dans la figure 5.10a, les résultats en fonction des valeurs de N montrent que la valeur maximale de m est 2. Ce résultat provient du coût très important en logique du sélecteur, ce coût provient du fait que les multiplexeurs sont implémentés avec des blocs logiques standard. S'il était possible d'avoir des éléments de base multiplexeurs, ce coût serait moins important.

La figure 5.10b présente un graphique montrant la valeur de m maximale en fonction de N. Dans ce graphique, on constate que pour des petites valeurs de N, le sélecteur n'est pas utile. Cependant, pour $N \geq 80$, on obtient $m \leq 10$. Cela implique que pour des jeux d'expressions régulières dont chacun des états du NFA résultant requiert moins de dix états précédents, le sélecteur peut être implémenté et réduit l'utilisation de mémoire. Enfin, ce graphique montre que m croît de façon monotone à mesure que N croît.

$$\begin{bmatrix} 3 \times m + 1 + \sum_{k=N-m+2}^{N+1} \lceil k + \log_2(k) \rceil \\ C \times (m + \sum_{k=N-m+2}^{N+1} \lceil \log_2(k) \rceil) \end{bmatrix} = \begin{bmatrix} 3 \times N + 1 \\ C \times N \end{bmatrix} \tag{5.7}$$

N	m
1-2	0
3-14	1
15-300	2

(a) Logique



(b) Mémoire

Figure 5.10 Valeurs maximales de m en fonction de N pour l'utilisation du sélecteur, (a) pour la logique et (b) pour la mémoire

5.3.3 Déterminer le nombre de RSPE de chaque type

Le nombre total de ERDE disponibles détermine la plus grande expression régulière qui peut être supportée par le système. Par ailleurs, les ERDE de type 1 sont suffisants pour implémenter n'importe quelle expression, mais plusieurs peuvent être requis dans le cas de la validation de listes de caractères. Comme l'architecture doit permettre le support de la plus

grande expression du jeu d'expression, il est essentiel de déterminer le nombre de ERDE de chaque type à implémenter.

Les valeurs optimales pour N_{T1} et N_{T2} dépendent donc des expressions qui doivent être supportées. Ce choix peut également permettre de simplifier le travail du synthétiseur. En effet, si lors de l'implémentation tous les éléments pour la validation des jetons sont intégrés alors aucune optimisation n'est nécessaire. Prenons par exemple le cas de l'expression `a[bf][cde]h`, une implémentation simple est d'avoir deux ERDE de type 1 et deux de type 2. Si seulement un ERDE de type 2 est disponible, il faut au minimum quatre ERDE de type 1. Dans le cas ou seulement 1 ERDE de type 2 est utilisable, alors il faut déterminer lequel des jetons `[bf]` ou `[cde]` doit être implémenté en utilisant des types 1.

5.3.4 Méthodes de modification de contexte

Cette section présente deux méthodes de reconfiguration de contextes : une en interrompant le système, l'autre pendant son fonctionnement.

La première méthode de reconfiguration nécessite d'arrêter le système pendant la modification d'un contexte. Dans cette méthode, les mémoires de contextes peuvent être des mémoires à simple port. Cette méthode simplifie également la gestion de reconfiguration, car il n'est pas nécessaire de vérifier que le contexte actuellement reconfiguré n'est pas le contexte actif. Cependant, l'arrêt du système peut s'avérer impossible notamment pour assurer la disponibilité de service.

La seconde méthode de reconfiguration assure la disponibilité du système. Pour fonctionner, cette méthode requiert l'utilisation de mémoires à double port. La reconfiguration en fonctionnement implique de protéger le contexte en cours de reconfiguration. Cela peut être fait de plusieurs manières. Une première façon est logicielle, elle consiste à empêcher les requêtes sur le contexte en cours de modification. Une seconde solution requiert l'ajout d'un contexte supplémentaire. Lors de la reconfiguration, ce dernier contexte est modifié. Une fois le contexte configuré, le numéro de l'ancien contexte est remplacé par le nouveau dans le système en amont.

Ces deux méthodes de configuration doivent donc être choisies en fonction des contraintes propres du système et de l'application visée. Une première requiert un arrêt du système alors que l'autre utilise plus de logique. Quoi qu'il en soit, ces deux méthodes prennent un avantage certain de la réduction du nombre de bits de configuration par rapport à des architectures à grains fins.

5.4 Résultats d'implémentation

Cette section présente les résultats d'implémentation des différents éléments de l'architecture et de l'architecture globale. Dans l'implémentation réalisée, $m = N$, il n'y a donc pas de *Sélecteur*. Trois métriques sont considérées : le nombre de LUT, le nombre de BM et la période d'horloge. Les mesures ont été faites après le placement et le routage pour un FPGA Xilinx Zynq-7 7z045 avec la suite d'outils Vivado 2015.4.

5.4.1 Bloc de validation d'état

Le tableau 5.1 présente les résultats d'implémentation ainsi que des valeurs théoriques estimées à l'aide de l'équation 5.3 pour le bloc *StateVal* en fonction de différentes valeurs de m et C. Ces résultats montrent tout d'abord la précision de la fonction d'estimation du coût pour les LUT. Comme l'implémentation est effectuée dans un FPGA, l'utilisation des mémoires n'est pas linéaire, il faut donc prendre cela en compte pour optimiser l'utilisation de cette ressource.

Comme attendu par la théorie, la consommation de logique ne dépend pas du nombre de contextes C mais uniquement du nombre d'états m. Le type de mémoire utilisé, à simple ou à double port, n'affecte pas la quantité de ressources utilisées. La période d'horloge est cependant très sensible à la variation de m. Cette variation s'explique, car le délai est principalement dû au temps de propagation du signal dans les fils. Enfin, on remarque que l'équation 5.3 est valide.

Tableau 5.1 Résultats d'implémentation du bloc *StateVal*

	$C = 512$			$C = 1024$		
	$m = 50$	$m = 116$	$m = 200$	$m = 50$	$m = 116$	$m = 200$
LUT Mesurées	30	70	120	30	70	120
LUT Calculées	30	70	120	30	70	120
BRAM simple port	1	2	3	1,5	3,5	6
BRAM double port	1	2	3	1,5	3,5	6
Période	3,8 ns	4 ns	4,2 ns	3,8 ns	4 ns	4,3 ns

5.4.2 Implémentation des ERDE

Cette section s'intéresse aux résultats d'implémentation des deux types de ERDE. Chacune des implémentations a été effectuée pour $m = 50$ et $m = 116$ et pour $C = 512$ et $C = 1024$. Les mémoires utilisées étaient des mémoires à double port.

ERDE de type 1 : Le tableau 5.2 montre les ressources consommées pour l'implémentation du type 1 ainsi que les périodes d'horloges minimales. L'implémentation du bloc de validation de caractère de ce type de ERDE donnait l'utilisation de 3 LUT. On devrait donc avoir un nombre de LUT proche du nombre nécessaire pour le bloc de validation d'état plus 3. Ces résultats sont cependant totalement différents de cette valeur attendue. Pour $m = 50$, le résultat attendu était de $30 + 3 = 33$ LUT, mais l'implémentation en nécessite 45. Lors de la comparaison des circuits générés après la synthèse, on remarque une grande différence entre la structure obtenue pour le bloc de validation d'états pour les cas où il est implémenté seul et le cas où il est implémenté au sein d'un ERDE. Dans le second cas, des parties de l'implémentation de la validation d'états sont implémentées avec des LUT de moins de six entrées. La période d'horloge n'est pas particulièrement impacté par ce changement ainsi que l'utilisation des mémoires. Enfin l'utilisation de logique est toujours indépendante du nombre de contextes.

Tableau 5.2 Résultats d'implémentation d'un ERDE de type 1

	$m = 50$			$m = 116$		
C	LUT	BRAM	Période	LUT	BRAM	Période
512	45	1,5	3,7 ns	92	2,5	4,0 ns
1024	45	2	3,7 ns	92	4	4,0 ns

ERDE de type 2 : Le tableau 5.3 présente les résultats d'implémentation d'un ERDE de type 2. L'implémentation du bloc de validation de caractère seul nécessite 69 LUT et 49 multiplexeurs 2 vers 1. Lors de l'implémentation du ERDE, pour $m = 50$ 50 multiplexeur 2 vers 1 sont utilisé, et pour $m = 116$ 48 multiplexeurs 2 vers 1 sont utilisés. Ce résultat provient d'un changement d'organisation effectué par le synthétiseur. Les résultats montrent que doubler le nombre de contextes double presque le nombre de BRAM. La période n'est par contre pas influencée par cette augmentation. La période est cependant influencée par la valeur de m, elle passe de 4,1 à 4,3 ns pour m allant de 50 à 116. Le nombre de LUT passe de 112 à 162 pour m passant de 50 à 116, cette valeur n'est cependant pas influencée par le nombre de contextes.

Tableau 5.3 Résultats d'implémentation d'un ERDE de type 2

	$m = 50$				$m = 116$			
C	LUT	MUX 2-1	BRAM	Période	LUT	MUX 2-1	BRAM	Période
512	112	50	5	4,1 ns	162	48	6	4,3 ns
1024	112	50	9	4,1 ns	162	48	11	4,3 ns

5.4.3 Système complet

Deux implémentations de l'architecture intermédiaire ont été réalisées. Chacune d'elles implémente cinq ERDE de type 2 et 111 de type 1, ce choix a été fait après une analyse des règles de Snort que le système peut supporter. La première implémentation a été faite pour 512 contextes et la seconde pour 1024. Les résultats intègrent également la logique pour gérer le flux de reconfiguration des contextes. La reconfiguration se fait en envoyant à chaque ERDE les éléments de configurations suite à la dé-sérialisation du flux de configuration.

Les résultats d'implémentation obtenus sont exposés dans le tableau 5.4. Le nombre de bascules se sépare en deux parties : une première pour la gestion du flux de configuration, la seconde pour les bascules nécessaires à la représentation d'un état. Le coût en logique est peu sensible au nombre de contextes, le nombre de LUT a même diminué en passant de 11 895 pour 512 contextes à 11 867 pour 1 024 contextes. Ce changement semble être majoritairement lié à l'optimisation du synthétiseur. Le nombre de blocs mémoires augmente dans les proportions attendues par rapport aux résultats d'implémentation individuelle des ERDE. Toutefois, la période augmente est n'est jamais inférieur à 5ns. Cette dernière passe de 5,5 ns pour 512 contextes à 6,3 ns pour 1024 contextes.

Tableau 5.4 Résultats d'implémentation de l'architecture intermédiaire

C	LUT	Mux 2-1	Bascules D	BRAM	Période
512	11 895	240	548	307,5	5,5 ns
1024	11 867	240	549	499	6,3 ns

5.5 Discussion

Ce chapitre a présenté la réalisation d'une architecture reconfigurable à gros grains. Cette architecture supporte jusqu'à 1024 contextes ce qui permet de supporter 1024 expressions régulières différentes. Le changement de contexte s'effectue en un seul cycle d'horloge, ce qui permet de supporter le pré-filtre intégré dans le fonctionnement de Snort. Grâce à l'utilisation de gros grains, le temps de modification d'un contexte et donc de mise à jour est plus court.

Divyasree et al. [44] ont aussi proposés des blocs reconfigurables pour la recherche d'expressions régulières. Toutefois, le grand temps de reconfiguration de ces blocs rend possible une attaque par déni de service. Par exemple, il serait possible d'envoyer du trafic qui requiert le changement continuel des configurations. Comme les données continuent d'arriver sur le

réseaux, une surcharge du système devient possible. Ce problème est solutionné dans la réalisation présentée dans ce chapitre en effectuant le changement de contexte en un seul cycle d'horloge.

De plus, la solution proposée par Divyasree et al. requiert plusieurs blocs pour effectuer la comparaison de plage de caractères. Ce problème est solutionné par l'architecture introduite dans ce chapitre grâce au ERDE de type 2.

L'architecture proposée dans ce chapitre requiert toutefois une grande quantité de mémoire interne au FPGA. Toutefois, une partie de la mémoire nécessaire pourrait être réduite par l'utilisation d'un sélecteur. L'utilisation d'un sélecteur nécessite l'utilisation de beaucoup de ressources dans les FPGA actuels, mais celles-ci pourraient être réduites si, par exemple, des modules d'interconnections étaient disponibles. Une solution multi-contexte proposée par Qu et al. [36] utilise une mémoire externe pour le stockage des contextes. Cette solution utilise le fait que plusieurs expressions régulières sont recherchées dans un flux de données pour masquer le temps de changement de contexte. Toutefois, une telle solution ne supporte pas efficacement le pré-filtre de Snort.

Les solutions proposées par Vasiliadis et al. [39] et par Wang et al. [40] basées sur des GPU permettent également l'utilisation du pré-filtre. Ces solutions ne permettent pas un traitement dans la continuité du réseau. En effet, les GPU utilisent un bus de communication qui s'avère peu efficace pour le transfert fréquent de petites quantités de données. Ces solutions requièrent donc de traiter les données après en avoir reçu un nombre important.

Le travail présenté ici valide des formules d'estimation pour le coût en ressources. Ces formules permettent de rapidement déterminer, selon les besoins de l'utilisateur, si la solution proposée peut être implémentée en fonction des ressources disponibles.

Finalement, l'utilisation d'une architecture intermédiaire donne deux niveaux de reconfiguration. Le premier est la mise à jour des expressions régulières supportées. Le second est la possibilité de synthétiser l'architecture de nouveau avec de nouveaux paramètres ainsi que de nouveaux modules. Il est ainsi possible d'adapter plus finement l'architecture utilisée en fonction des besoins. Par exemple, d'autres modules tel que des compteurs pourraient être ajoutés pour supporter une plus grande variété d'éléments du langage régulier.

CHAPITRE 6 CONCLUSION

Le travail présenté dans ce mémoire propose des architectures de recherche de caractères et plus précisément des solutions pour l'implémentation d'automates. Dans ce chapitre, un résumé du travail réalisé est proposé. Dans un premier temps une synthèse des réalisations est proposée. Dans une deuxième section, certaines limites du travail réalisé sont revues. Finalement des axes futurs de recherche sont exposés.

6.1 Synthèse des travaux

Trois solutions pour la recherche de texte ont été proposées dans ce travail. Une première implémente un automate déterministe afin de faire de la recherche de chaînes de caractères. La deuxième réalisation propose de compiler des expressions régulières et d'implémenter le NFA de ces dernières sur un FPGA. La troisième implémentation utilise le concept d'architecture intermédiaire pour implémenter le NFA résultant de la compilation d'une expression régulière.

L'implémentation de la recherche de chaînes de caractères démontre la capacité des FPGA à traiter des problèmes de recherche massive de donnés. Cette solution a également montré la liberté d'action que donne l'utilisation d'un FPGA pour ce type d'application. En effet, l'utilisation du FPGA permet d'adapter la représentation des nœuds dans le cas de l'automate à parcourir en fonction du jeu de données en entrée.

La réalisation du compilateur d'expressions régulières a démontré la possibilité d'automatisation du processus de réalisation d'un design pour un FPGA. Le processus proposé permet d'une part de réaliser l'architecture, mais également d'automatiser la génération des tests. Cette solution rend donc l'automatisation du processus entre la génération de la règle et l'utilisation du FPGA plus rapide.

La troisième solution propose d'implémenter une architecture à deux niveaux de configurations. Elle permet d'utiliser les mécanismes intégrés dans Snort pour rendre la recherche d'expressions régulières plus efficace. Cela se fait en utilisant le concept des architectures intermédiaires. Cette solution permet à la fois de supporter les mises à jour de règles, mais permet également d'intégrer des mises à jour systèmes. Dans cette réalisation, des méthodes d'approximation de coût ont été proposées afin de déterminer d'une part les bons choix de répartition pour les modules, mais également estimer les ressources nécessaires pour l'implémentation de l'architecture. Cette réalisation a résulté en la publication d'un article à la conférence ACM Great Lakes Symposium on VLSI 2017 (GLSVLSI'17) [17].

6.2 Limitations des réalisations effectuées

L'implémentation de la recherche de chaînes de caractère sur le FPGA possède deux limitations précédemment exprimées. La première est la gestion des mises à jour. Celle-ci nécessite d'approfondir des solutions d'implémentations. La seconde est également liée aux mises à jour, mais du nombre de règles. Le système est en effet très sensible à cette dernière puisqu'un bit est nécessaire par règle. Tant que le nombre de règles ne change pas, la solution est efficace, mais l'ajout d'une règle peut nécessiter une nouvelle implémentation. Une solution consiste à prévoir ce nouveau nombre de règles, mais cela peut augmenter l'utilisation de ressources et peut s'avérer non nécessaire.

Le compilateur permet de grandement automatiser le processus d'implémentation, mais souffre de la nécessité de refaire la synthèse pour chaque mise jour. Ceci peut être un point critique, notamment dans les systèmes de sécurité, car le support de la nouvelle règle devient un processus long. De plus, comme il faut reconfigurer le FPGA, il est nécessaire d'interrompre le traitement ou d'avoir une redondance du système pour prendre le relais. Un travail futur pour résoudre ce problème serait l'intégration de l'utilisation de la reconfiguration partielle.

L'architecture intermédiaire permet de partiellement résoudre le problème de mise à jour du compilateur. Il le résout, car il est possible de facilement reconfigurer un contexte. La résolution n'est que partielle pour deux cas de figure, la nouvelle expression requiert trop d'états ou bien si une nouvelle expression doit être ajoutée et que le système les utilise toutes. Le premier problème qui est le support d'une expression plus longue possède deux axes pour le résoudre. Un premier axe est la décomposition de l'automate en plusieurs automates. Un second axe rejoint l'axe de réflexion pour le second problème. Une solution est de refaire la synthèse de l'architecture intermédiaire. La reconfiguration partielle serait un élément intéressant, mais son exploitation semblerait être un objectif ambitieux étant donné la dépendance entre les modules. La deuxième grande limitation de l'architecture proposée est le débalancement entre la consommation de mémoire par rapport à l'utilisation de logique. Ceci implique la non-utilisation d'une grande partie du FPGA ce qui peut rendre le système coûteux.

Finalement, l'architecture intermédiaire et le compilateur sont limités dans le langage des expressions régulières supporté. Une amélioration future est donc d'ajouter le support de plus d'éléments du langage régulier, puis d'étendre le support au langage non régulier tel que PCRE.

6.3 Travaux futurs

La section 6.2 présente certaines limitations des différentes réalisations et propose certains axes de réflexion. Dans cette section, les axes de réflexion précédents sont résumés et d'autres axes de recherche plus généraux sont proposés.

Un axe majeur d'amélioration est l'utilisation de la reconfiguration partielle des FPGA. Celle-ci permettrait dans de nombreux cas de figure de réduire les temps de mise à jour ainsi que de simplifier ces dernières. Cet axe se décompose en deux sous-axes : un premier, relié à la technologie, est de réduire le temps de transfert des flux de configuration ; le second est de simplifier la réalisation de configuration partielle. À l'heure actuelle, celle-ci requiert une division de l'architecture dans des régions du FPGA, des outils simplifiant sa mise en œuvre seraient bienvenus.

Plus généralement, les architectures intermédiaires permettent un bon compromis entre temps de développement et intégration sur le FPGA. La difficulté est de trouver des solutions permettant de supporter une plus grande variété d'applications. Si de tels systèmes se démocratisent, il sera également intéressant de chercher des solutions pour gérer les interconnexions. Finalement, ces architectures ouvrent la possibilité de proposer des accélérateurs plus rapidement configurables et ainsi de combiner ceux-ci avec l'exécution d'applications sur un processeur à usage général.

RÉFÉRENCES

[1] La naissance du web, CERN. adresse : https://home.cern/fr/topics/birth-web.

[2] N. K. Jha, "Internet-of-Medical-Things", in *Proceedings of the on Great Lakes Symposium on VLSI 2017*, sér. GLSVLSI '17, Banff, Alberta, Canada : ACM, 2017, p. 7–7, ISBN : 978-1-4503-4972-7. DOI : 10.1145/3060403.3066861. adresse : http://doi.acm.org/10.1145/3060403.3066861.

[3] A. Greenberg, J. Hamilton, D. A. Maltz et P. Patel, "The Cost of a Cloud : Research Problems in Data Center Networks", *SIGCOMM Comput. Commun. Rev.*, t. 39, nº 1, p. 68–73, déc. 2008, ISSN : 0146-4833. DOI : 10.1145/1496091.1496103. adresse : http://doi.acm.org/10.1145/1496091.1496103.

[4] "5G white paper", rapp. tech. adresse : https://www.ngmn.org/uploads/media/NGMN_5G_White_Paper_V1_0.pdf.

[5] "Software-Defined Networking : The New Norm for Networks", rapp. tech. adresse : https://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf.

[6] P. Bosshart, D. Daly, M. Izzard, N. McKeown, J. Rexford, D. Talayco, A. Vahdat, G. Varghese et D. Walker, "Programming Protocol-Independent Packet Processors", *CoRR*, t. abs/1312.1719, 2013. adresse : http://arxiv.org/abs/1312.1719.

[7] COP21, "Accord de Paris", déc. 2015. adresse : http://unfccc.int/files/essential_background/convention/application/pdf/french_paris_agreement.pdf.

[8] W. V. Heddeghem, S. Lambert, B. Lannoo, D. Colle, M. Pickavet et P. Demeester, "Trends in worldwide ICT electricity consumption from 2007 to 2012", *Computer Communications*, t. 50, p. 64–76, 2014, Green Networking, ISSN : 0140-3664. DOI : 10.1016/j.comcom.2014.02.008. adresse : http://www.sciencedirect.com/science/article/pii/S0140366414000619.

[9] J. Koomey, "Growth in data center electricity use 2005 to 2010", 2011.

[10] K. J. S. Hoo, *How Much Is Enough ? A Risk-Management Approach to Computer Security*. adresse : https://cisac.fsi.stanford.edu/sites/default/files/soohoo.pdf.

[11] V. Paxson, "Bro : a System for Detecting Network Intruders in Real-Time", *Computer Networks*, t. 31, nº 23-24, p. 2435–2463, 1999. adresse : http://www.icir.org/vern/papers/bro-CN99.pdf.

[12] SNORT Network Intrusion Detection System. adresse : http://www.snort.org.

[13] H. Zimmermann, "OSI Reference Model - The ISO Model of Architecture for Open Systems Interconnection", *IEEE Transactions on Communications*, t. 28, nº 4, p. 425–432, 1980, ISSN : 0090-6778. DOI : 10.1109/TCOM.1980.1094702.

[14] I. Ion, N. Sachdeva, P. Kumaraguru et S. Čapkun, "Home is Safer Than the Cloud !: Privacy Concerns for Consumer Cloud Storage", in *Proceedings of the Seventh Symposium on Usable Privacy and Security*, sér. SOUPS '11, Pittsburgh, Pennsylvania : ACM, 2011, 13 :1–13 :20, ISBN : 978-1-4503-0911-0. DOI : 10.1145/2078827.2078845. adresse : http://doi.acm.org/10.1145/2078827.2078845.

[15] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao et D. Burger, "A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services", *Commun. ACM*, t. 59, nº 11, p. 114–122, oct. 2016, ISSN : 0001-0782. DOI : 10.1145/2996868. adresse : http://doi.acm.org/10.1145/2996868.

[16] P. K. Gupta, "Xeon+ fpga platform for the data center", in *Fourth Workshop on the Intersections of Computer Architecture and Reconfigurable Logic*, t. 119, 2015.

[17] T. Luinaud, Y. Savaria et J. P. Langlois, "An FPGA Coarse Grained Intermediate Fabric for Regular Expression Search", in *Proceedings of the on Great Lakes Symposium on VLSI 2017*, sér. GLSVLSI '17, Banff, Alberta, Canada : ACM, 2017, p. 423–426, ISBN : 978-1-4503-4972-7. DOI : 10.1145/3060403.3060429. adresse : http://doi.acm.org/10.1145/3060403.3060429.

[18] N. Chomsky, "On certain formal properties of grammars", *Information and Control*, t. 2, nº 2, p. 137–167, 1959, ISSN : 0019-9958. DOI : 10.1016/S0019-9958(59)90362-6. adresse : http://www.sciencedirect.com/science/article/pii/S0019995859903626.

[19] S. C. Kleene, "Representation of events in nerve nets and finite automata", DTIC Document, rapp. tech., 1951.

[20] A. V. Aho, M. S. Lam, R. Sethi et J. D. Ullman, *Compilers : Principles, Techniques, and Tools (2Nd Edition)*, English, 2nd –. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 2006, ISBN : 0321486811.

[21] A. Salomaa, *Theory of Automata*, sér. International Series in Pure and Applied Mathematics. Pergamon Press, 1969.

[22] J. Sakarovitch, *Eléments de théorie des automates*. Vuibert informatique, 2003.

[23] A. V. Aho et M. J. Corasick, “Efficient String Matching : An Aid to Bibliographic Search”, *Commun. ACM*, t. 18, nº 6, p. 333–340, juin 1975, ISSN : 0001-0782. DOI : 10.1145/360825.360855. adresse : http://doi.acm.org/10.1145/360825.360855.

[24] N. Tuck, T. Sherwood, B. Calder et G. Varghese, “Deterministic memory-efficient string matching algorithms for intrusion detection”, in *IEEE INFOCOM 2004*, t. 4, 2004, 2628–2639 vol.4. DOI : 10.1109/INFCOM.2004.1354682.

[25] A. Bremler-Barr, Y. Harchol et D. Hay, “Space-time tradeoffs in software-based deep Packet Inspection”, in *2011 IEEE 12th International Conference on High Performance Switching and Routing*, 2011, p. 1–8. DOI : 10.1109/HPSR.2011.5985996.

[26] D. R. Morrison, “PATRICIA — Practical Algorithm To Retrieve Information Coded in Alphanumeric”, *J. ACM*, t. 15, nº 4, p. 514–534, oct. 1968, ISSN : 0004-5411. DOI : 10.1145/321479.321481. adresse : http://doi.acm.org/10.1145/321479.321481.

[27] X. J. A. Bellekens, C. Tachtatzis, R. C. Atkinson, C. Renfrew et T. Kirkham, “A Highly-Efficient Memory-Compression Scheme for GPU-Accelerated Intrusion Detection Systems”, in *Proceedings of the 7th International Conference on Security of Information and Networks*, sér. SIN ’14, Glasgow, Scotland, UK : ACM, 2014, 302 :302–302 :309, ISBN : 978-1-4503-3033-6. DOI : 10.1145/2659651.2659723. adresse : http://doi.acm.org/10.1145/2659651.2659723.

[28] C. H. Lin, S. Y. Tsai, C. H. Liu, S. C. Chang et J. M. Shyu, “Accelerating String Matching Using Multi-Threaded Algorithm on GPU”, in *2010 IEEE Global Telecommunications Conference GLOBECOM 2010*, 2010, p. 1–5. DOI : 10.1109/GLOCOM.2010.5683320.

[29] A. B. Lacroix, J. P. Langlois, F.-R. Boyer, A. Gosselin et G. Bois, “Node Configuration for the Aho-Corasick Algorithm in Intrusion Detection Systems”, in *Proceedings of the 2016 Symposium on Architectures for Networking and Communications Systems*, sér. ANCS ’16, Santa Clara, California, USA : ACM, 2016, p. 121–122, ISBN : 978-1-4503-4183-7. DOI : 10.1145/2881025.2889473. adresse : http://doi.acm.org/10.1145/2881025.2889473.

[30] L. Tan et T. Sherwood, “A High Throughput String Matching Architecture for Intrusion Detection and Prevention”, *SIGARCH Comput. Archit. News*, t. 33, nº 2, p. 112–122, mai 2005, ISSN : 0163-5964. DOI : 10.1145/1080695.1069981. adresse : http://doi.acm.org/10.1145/1080695.1069981.

[31] P. Piyachon et Y. Luo, “Efficient Memory Utilization on Network Processors for Deep Packet Inspection”, in *Proceedings of the 2006 ACM/IEEE Symposium on Architecture for Networking and Communications Systems*, sér. ANCS ’06, San Jose, California, USA : ACM, 2006, p. 71–80, ISBN : 1-59593-580-0. DOI : 10.1145/1185347.1185358. adresse : http://doi.acm.org/10.1145/1185347.1185358.

[32] K. Thompson, “Programming Techniques : Regular Expression Search Algorithm”, *Commun. ACM*, t. 11, nº 6, p. 419–422, juin 1968, ISSN : 0001-0782. DOI : 10.1145/363347.363387. adresse : http://doi.acm.org/10.1145/363347.363387.

[33] R. W. Floyd et J. D. Ullman, “The Compilation of Regular Expressions into Integrated Circuits”, *J. ACM*, t. 29, nº 3, p. 603–622, juil. 1982, ISSN : 0004-5411. DOI : 10.1145/322326.322327. adresse : http://doi.acm.org/10.1145/322326.322327.

[34] R. Sidhu et V. K. Prasanna, “Fast Regular Expression Matching Using FPGAs”, in *Field-Programmable Custom Computing Machines, 2001. FCCM ’01. The 9th Annual IEEE Symposium on*, 2001, p. 227–238. adresse : http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1420919.

[35] A. Mitra, W. Najjar et L. Bhuyan, “Compiling PCRE to FPGA for Accelerating SNORT IDS”, in *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for Networking and Communications Systems*, sér. ANCS ’07, Orlando, Florida, USA : ACM, 2007, p. 127–136, ISBN : 978-1-59593-945-6. DOI : 10.1145/1323548.1323571. adresse : http://doi.acm.org/10.1145/1323548.1323571.

[36] Y. Qu, Y. H. E. Yang et V. K. Prasanna, “Large-scale multi-flow regular expression matching on FPGA”, in *2012 IEEE 13th International Conference on High Performance Switching and Routing*, 2012, p. 70–75. DOI : 10.1109/HPSR.2012.6260830.

[37] A. C. Mihal, C. Sauer et K. Keutzer, “Designing a Sub-RISC Multi-Gigabit Regular Expression Processor”, 2006.

[38] A. Mihal, S. Weber et K. Keutzer, “Chapter 13 - Sub-RISC Processors”, in *Customizable Embedded Processors*, sér. Systems on Silicon, P. Lenne et R. Leupers, éds., Burlington : Morgan Kaufmann, 2007, p. 303–336. DOI : 10.1016/B978-012369526-0/50014-0. adresse : http://www.sciencedirect.com/science/article/pii/B9780123695260500140.

[39] G. Vasiliadis, M. Polychronakis, S. Antonatos, E. P. Markatos et S. Ioannidis, “Regular Expression Matching on Graphics Hardware for Intrusion Detection”, in *Recent Advances in Intrusion Detection : 12th International Symposium, RAID 2009, Saint-Malo,*

France, September 23-25, 2009. Proceedings, E. Kirda, S. Jha et D. Balzarotti, éds. Berlin, Heidelberg : Springer Berlin Heidelberg, 2009, p. 265–283, ISBN : 978-3-642-04342-0. DOI : 10.1007/978-3-642-04342-0_14. adresse : http://dx.doi.org/10.1007/978-3-642-04342-0_14.

[40] L. Wang, S. Chen, Y. Tang et J. Su, "Gregex : GPU Based High Speed Regular Expression Matching Engine", in *2011 Fifth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, 2011, p. 366–370. DOI : 10.1109/IMIS.2011.107.

[41] J. van Lunteren et A. Guanella, "Hardware-accelerated regular expression matching at multiple tens of Gb/s", in *2012 Proceedings IEEE INFOCOM*, 2012, p. 1737–1745. DOI : 10.1109/INFCOM.2012.6195546.

[42] J. van Lunteren, "High-Performance Pattern-Matching for Intrusion Detection", in *Proceedings IEEE INFOCOM 2006. 25TH IEEE International Conference on Computer Communications*, 2006, p. 1–13. DOI : 10.1109/INFOCOM.2006.204.

[43] J. V. Lunteren, C. Hagleitner, T. Heil, G. Biran, U. Shvadron et K. Atasu, "Designing a Programmable Wire-Speed Regular-Expression Matching Accelerator", in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, sér. MICRO-45, Vancouver, B.C., CANADA : IEEE Computer Society, 2012, p. 461–472, ISBN : 978-0-7695-4924-8. DOI : 10.1109/MICRO.2012.49. adresse : http://dx.doi.org/10.1109/MICRO.2012.49.

[44] J. Divyasree, H. Rajashekar et K. Varghese, "Dynamically reconfigurable regular expression matching architecture", in *2008 International Conference on Application-Specific Systems, Architectures and Processors*, 2008, p. 120–125. DOI : 10.1109/ASAP.2008.4580165.

[45] N. W. Bergmann, S. K. Shukla et J. Becker, "QUKU : A Dual-layer Reconfigurable Architecture", *ACM Trans. Embed. Comput. Syst.*, t. 12, n° 1s, 63 :1–63 :26, mar. 2013, ISSN : 1539-9087. DOI : 10.1145/2435227.2435259. adresse : http://doi.acm.org/10.1145/2435227.2435259.

[46] J. Coole et G. Stitt, "Intermediate Fabrics : Virtual Architectures for Circuit Portability and Fast Placement and Routing", in *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, sér. CODES/ISSS '10, Scottsdale, Arizona, USA : ACM, 2010, p. 13–22, ISBN : 978-1-60558-905-3. DOI : 10.1145/1878961.1878966. adresse : http://doi.acm.org/10.1145/1878961.1878966.

[47] R. Lysecky, K. Miller, F. Vahid et K. Vissers, “Firm-core virtual FPGA for just-in-time FPGA compilation”, in *ACM/SIGDA International Symposium on Field Programmable Gate Arrays - FPGA*. 2005, p. 271.

[48] A. Brant et G. G. F. Lemieux, “ZUMA : An Open FPGA Overlay Architecture”, in *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, 2012, p. 93–96. DOI : 10.1109/FCCM.2012.25.

[49] S. Vakili, J. P. Langlois, B. Boughzala et Y. Savaria, “Memory-Efficient String Matching for Intrusion Detection Systems Using a High-Precision Pattern Grouping Algorithm”, in *Proceedings of the 2016 Symposium on Architectures for Networking and Communications Systems*, sér. ANCS ’16, Santa Clara, California, USA : ACM, 2016, p. 37–42, ISBN : 978-1-4503-4183-7. DOI : 10.1145/2881025.2881031. adresse : http://doi.acm.org/10.1145/2881025.2881031.

[50] A. V. Aho, M. S. Lam, R. Sethi et J. D. Ullman, *Compilers : Principles, Techniques, and Tools (2Nd Edition)*, English, 2nd –. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 2006, chap. 1, p. 1, ISBN : 0321486811.

[51] K. Compton et S. Hauck, “Reconfigurable Computing : A Survey of Systems and Software”, *ACM Comput. Surv.*, t. 34, n° 2, p. 171–210, juin 2002, ISSN : 0360-0300. DOI : 10.1145/508352.508353. adresse : http://doi.acm.org/10.1145/508352.508353.

[52] R. Hartenstein, “Coarse grain reconfigurable architectures”, in *Proceedings of the ASP-DAC 2001. Asia and South Pacific Design Automation Conference 2001 (Cat. No.01EX455)*, 2001, p. 564–569. DOI : 10.1109/ASPDAC.2001.913368.

[53] Xilinx, *7 Series FPGAs Data Sheet : Overview*. adresse : https://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf.